UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Knowledge-based Verification of Service Compositions

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

in der

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von Sven Walther am

27. Juli 2017

Abstract

As of today, modern software is already provided as-a-service and no longer sold as monolithic application. Tomorrow, with visions as on-the-fly computing, these services will be created automatically, composed from existing services according to the request of a user, based in an actual business domain. Therefore, *correctness* of such a composition has to be guaranteed on the fly, as well.

Verifying software in an on-the-fly context leads to a major challenge: It can only yield results if it is rooted in formal specification. At the same time, verification meets the domains of service description, composition, and knowledge modeling, all using different formalisms with different semantics. Therefore, we need a framework which enables domain knowledge modeling as well as service and composition specification, while at the same time using a shared theoretical foundation to enable formal verification.

This thesis provides such a framework. It uses predicate logic as a common ground to build an integrated framework to model knowledge, use this knowledge to create service descriptions, and to define a workflow, or composition, language. Additionally, it lifts compositions to templates, and provides a semantics which is parameterized not only in the logical structures, but also in the possible instantiations of templates in arbitrary business domains.

It also provides a sound and complete proof calculus to show correctness of templates. Additionally, it leverages the use of the framework in an on-the-fly context by replacing the need of composition verification with a check of side conditions of a template during instantiation, and by providing an encoding of correctness checks to make use of automatic verification using satisfiability solvers.

Zusammenfassung

Software wird heute als Dienstleistung, oder *Software-as-a-Service*, angeboten und nicht mehr als monolithische Anwendung. In Visionen wie dem On-The-Fly Computing werden diese Dienste (Services) automatisch erzeugt, zusammengestellt auf Basis von konkreten Nutzeranforderungen innerhalb einer Anwendungsdomäne. Das bedeutet, dass auch die *Korrektheit* solcher komponierten Dienste automatisch und "on-the-fly" sichergestellt werden muss.

Software-Verifikation "on-the-fly" durchzuführen ist eine besondere Herausforderung, da sie nur dann Resultate erzeugt, wenn sie auf formalen Spezifikationen arbeitet. Im On-The-Fly-Kontext arbeitet sie gleichzeitig auf den Bereichen von Service-Spezifikation, Komposition und Wissensmodellierung, die jeweils ihre eigenen Formalismen mit unterschiedlichen Semantiken mitbringen. Daher benötigen wir ein Gesamtsystem, das uns einerseits die Modellierung von domänenspezifischem Wissen, Diensten und Kompositionen ermöglicht und andererseits die theoretischen Grundlagen für eine formale Verifikation bereitstellt.

Die vorliegende Arbeit bietet ein solches Gesamtsystem. Auf Basis von Prädikatenlogik ermöglicht sie die Modellierung von Domänenwissen und nutzt genau dieses Wissen, um Dienste zu spezifieren und eine Kompositions- oder Ablaufsprache zu definieren. Zusätzlich ermöglicht sie die Beschreibung von Vorlagen (Templates) und definiert eine Semantik, die einerseits durch logische Strukturen und andererseits durch mögliche Instanziierungen der Vorlagen zu Kompositionen innerhalb beliebiger Wissensdomänen parametriert ist.

Die Arbeit definiert außerdem einen Beweiskalkül zum Nachweis der Korrektheit von Vorlagen und zeigt seine Korrektheit und Vollständigkeit. Zusätzlich erleichtert sie die Anwendung des Systems im On-The-Fly-Kontext, indem sie zeigt, dass ein Korrektheitsnachweis einer Komposition auf Basis einer korrekten Vorlage durch den Nachweis von Nebenbedingungen im Rahmen der Instanziierung ersetzt werden kann. Darüber hinaus bietet sie eine logische Kodierung der Korrektheitseigenschaften, um eine automatische Verifikation auf Basis von Erfüllbarkeitsprüfern zu ermöglichen.

Acknowledgements

I would like to thank my advisor Prof. Dr. Heike Wehrheim for her support, guidance, and patience during my years at her group and beyond. I would also like to thank Prof. Dr. Hans Kleine Büning, Prof. Dr. Eyke Hüllermeier, Jun.-Prof. Dr. Michaela Geierhos, and Dr. Theo Lettmann for serving as thesis committee members.

Thanks go to my colleagues over the years: Thomas P. Ruhroth, Nils Timm, Daniel Wonisch, Alexander Schremmer, Dominik Steenken, Galina Besova, Steffen Ziegert, Tobias Isenberg, Marie-Christine Jakobs, Oleg Travkin, Steffen Beringer, Manuel Töws, and Elisabeth Schlatt. I thank all the colleagues of the CRC 901, esp. Matthias Becker, Marie Platenius, Svetlana Arifulina, and Felix Mohr. Thanks go also to Maryam Lexow, Andreas Krakau, and Siddhartha Moitra, whose theses and code explored some of the topics of this thesis in their early stages.

At last, I want to thank Saskia, who supported me during all the different stages of writing – and finishing – this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research tends to be specialized to produce new insights into complex topics. However, multidisciplinarity and cross-over research provide new results by mutual inspiration. While first successes may come from pragmatic approaches to combine different topics, on the long run a solid foundation is necessary. This thesis aims at providing such a foundation for three research topics of computer science, which are typically handled separately or in pairs:

(1) Formal knowledge modeling, where vocabulary and relations of a domain are modeled with mathematical formalisms;

(2) Service specification and composition, where software is delivered as a service and can be combined to new compositions due to its modularity;

(3) And program verification, where properties of a computer program are mathematically proved or disproved.

This chapter discusses the joined context of these topics as it can be found in the paradigm of *on-the-fly computing* (OTF computing). OTF computing comprises of the vision of a flexible (economic) market of software, where software services can be searched, combined, and delivered automatically. It gives rise not only to the motivation of joining the three topics in one area of application, but it also derives the need of a shared mathematical foundation, which lies at the heart of this thesis.

This context includes mathematical knowledge modeling as a necessary tool to leverage rich descriptions of the behavior of software services and paves the way for service composition. In software engineering, single software services are combined to service compositions with workflows in the sense of formally defined sequences of actions, or, in this case, services. The application of program verification – with the meaning as a rigorous proof technique and not as a validation by means of testing or simulation – as a technique to guarantee *correct* service compositions is a consequent result in this scenario.

The introduction discusses the overall scenario and motivates the need for a shared mathematical foundation of the three main topics, thus crystallizing the need of the contributions of this thesis.

1.1 A Theoretical Framework

Computer science, both as a science and in its applications, is in a constant, fast-paced evolution. One of the most recent changes includes software which is readily available everywhere, always, and in every possible configuration necessary. This includes two remarkable changes in paradigm: Software is no longer a monolithic program with as much specialized functionality as possible, and it is not purchased once and used until a user needs a more recent version.

Instead, software is modular. On a technical level, this is anything but new, as modularity is a basic part of any standard software design approach (e.g., Szyperski, 1998). But today, end users pay only for software they really need, and they no longer purchase it but rent it. This *Software as a Service* approach (SaaS, e.g., Armbrust et al., 2010) manifests itself in products like Microsoft's Office 365[1] or Adobe's Creative Cloud[2], where different tools or combinations thereof can be rented and used anywhere. On a smaller scale of complexity cloud services like Dropbox[3], Trello[4], Evernote[5], and numerous others deliver their services in different options or plans, typically with a free basic version and extra functionality for monthly fees. This development is recent, but in no way unexpected, as for business and technical users, computing services have been available before, e.g., the *Amazon Web Services* (AWS)[6].

Taking the idea of modular, remotely executed mainstream services further leads us to three core characteristics of software use:

- Modularity of services and as-a-service, or on-demand, use of software suggests having not only a few, but several service providers. Thus, there will be a heterogeneous *service market* with different roles: *Service providers* will create and provide services, others provide repositories, manage service discovery, execution, and other tasks.
- To discover a specific service on this market, it is necessary to specify the requirements to it, both functional and non-functional (e.g., mean runtime, cost, ...). To this end, a formal specification used for service description and, consequently, requirements description is necessary.

[1]Office suite, `https://www.office.com`, retrieved July 21, 2017.
[2]Media Solutions, `https://www.adobe.com/creativecloud.html`, retrieved July 21, 2017.
[3]Cloud storage service, `https://www.dropbox.com`, retrieved July 21, 2017.
[4]Whiteboard-style organizer, `https://trello.com`, retrieved July 21, 2017.
[5]Notebook-style organizer, `https://evernote.com`, retrieved July 21, 2017.
[6]Cloud infrastructure, `https://aws.amazon.com`, retrieved July 21, 2017.

- With modular services contrasting specific needs, it is improbable to find a fitting service when functional requirements get more complex. Thus, the need to combine services arises, where suitable existing services are found on the market and combined to a complex *service composition*.

The paradigm of *on-the-fly computing* (OTF computing) summarizes this vision. It includes several aspects: Economic dynamics on a heterogeneous market with all implications of trust, reputation, and privacy; formal needs to describe and merchandise services; distributed deployment and execution of composed services, and more (Happe et al., 2013).

A typical scenario works as follows:

(1) A user – business or end user – creates a formal specification of a service. This requirements specification is centered on a functional description of the expected behavior; thus, it describes the semantics of the requested service.

(2) The requirements specification is used by one or more on-the-fly *provider(s)* (OTF provider) to search the market for a matching service.

(3) If such a service is not found, the requirements specification serves as a goal for a *service composition* process. Here, services from the market are combined in a workflow to create a *service composition*, whose overall behavior matches the user's initial requirements.

(4) Depending on the composition techniques, the resulting composition is either correct by construction, or it has to be *verified* against the requirements in a separate step.

(5) If one or more verified compositions are found, one of them is selected. This may happen by non-functional requirements such as price or mean runtime, by some ranking function, or by the user.

(6) The selected composition (or service) is deployed on *compute centers* and executed. Depending on the exact composition, this may range from a one-time execution to a complex service that is hosted to be used by several end users.

In this thesis, we take a software engineering point of view and focus on step (4), *verification*. We restrict the context to the relevant topics to enable verification, namely service specification and composition. Thus, ignoring economic and deployment topics, we can distill three core requirements to this scenario:

- We need a means of *formal specification* of service behavior as well as requirements towards the same.

- Actual services are not abstract, but exist within a specific business domain, or domains. Thus, to obtain a functional specification, *knowledge* of this domain and its internal relations has to be incorporated into the service descriptions.
- To enable services tailored towards specific user requirements, the *composition* of services is essential. This way, existing services are used to create complex ones, ideally in an automatic fashion.

At the heart of these three requirements is *knowledge base modeling*. Originating from artificial intelligence research (Davis et al., 1993), it provides the tools to formally specify knowledge of a specific domain in terms of its vocabulary and the various relations between terms created from this vocabulary. Additionally, it provides a *semantics* for this knowledge. The semantics depend on the exact formalism used; In this thesis, we formalize knowledge using logic, thus we have semantics based on logical structures. There are other ways to capture domain knowledge, and not all of them require a formalization in a mathematical sense. Chapter 2 discusses different approaches of knowledge base modeling and introduces the formal foundation as used throughout this thesis.

While a formalized knowledge base can be used to reason about the knowledge within the modeled domain, its vocabulary can also be used as a means to describe the behavior of software. This includes not only “tagging” a software service with keywords, but first and foremost the specification in terms of *pre- and postconditions* of a service. This way, it precisely captures the requirements to the *input* of the service (the precondition) and the guarantees of its *output* (the postcondition). In contrast to syntactical service descriptions, e.g., in terms of method signatures accompanied with an informal, human-readable text (as in standard Application Programming Interfaces, or APIs), this approach enables a precise description of the *semantics* of the service.

In practice, software (or service) specification languages like the *Web Service Description Language* (WSDL) (Farrell and Lausen, 2007, Weerawarana et al., 2007) and the OWL-S ontology (Martin et al., 2005) combine a classical syntactic interface specification with pre- and postconditions. The latter uses expressions built on vocabulary defined using the *Web Ontology Language* (OWL, Patel-Schneider et al., 2012), a predominant knowledge base specification language with a logic semantics. Chapter 3 gives an overview on current approaches of service specification as well as combinations with knowledge base descriptions, before giving formal definitions of services for this thesis.

With formal behavior specifications of services in terms of pre- and postconditions, it is also possible to specify the requirements towards a service. In an OTF scenario, services are specialized, and requirements are specific to a certain business use case or end user. Therefore, it is unlikely that there always exists a service which matches existing requirements. In that case, the process of *service composition* is responsible to create a new, complex service to meet the requirements. To do so, the requirement specification is treated as a goal for a composition problem. Existing services satisfying the requirements partially (or enabling other services to do so) are combined in a workflow to create a new, complex service, or service composition. The behavior of this service, that is, its semantics, meets the requirements of the user.

Service (or software) composition is an active research topic of its own. Depending on the complexity of service descriptions, the structure of the workflow of a composition, and the requirements, the task of finding a composition may or may not be feasible. A prominent benchmark for composition approaches is the *Web Service Challenge* (Weise et al., 2014). Apart from the composition problem itself, the use of a *workflow* introduces a new aspect to the black box style description of service semantics. To the user, a composition is indistinguishable from a service, though it has no explicit pre- and postcondition to describe its semantics. Instead, its semantics result from its workflow and the services used in the composition. Therefore, the semantics of the workflow have to be compatible with the semantics of service – and requirements – specifications, and, consequently, with the semantics of a formal knowledge base.

Creating a service composition has the intention to create compositions that are *correct*, that is, that meet the original requirements. This works in some cases, but not in others. For example, when the composition process aims at compositions with a complex, non-sequential workflow, including conditional executions and loops, using a *workflow template* is one approach to find a service composition (Mohr and Walther, 2014). In that case, the template is defined manually by some domain expert, and the composition process finds services from a domain to replace "placeholders". Chapter 4 elaborates on templates formally. Another example is a composition process that makes simplifying assumptions about the complexity of pre- and postconditions, or the domain knowledge used to formalize them.

Whenever a composition is created with these techniques, correctness is not necessarily guaranteed by construction. Instead, it has to be proved in a separate step, step (4) in the introductory scenario. *Program verification* is the research topic which covers proof techniques to show that a certain program meets its specification or avoids error states (Clarke et al., 1994, Grumberg and Long, 1991). While workflows in a broader sense are a list of actions to be executed to reach a goal, in a computer science sense they are formally defined control structures, resembling programming languages. Conceptually, workflows and programming languages differ only in their area of application: While programming languages are used to program a computer directly, workflows control actions on an abstract level. Consequently, the difference between both depends often on the point of view. Chapter 3 discusses the history of workflows and their relation to programming languages in more detail. As workflows in a technical sense are control flow structures controlling atomic actions, being it services or program statements, techniques from program verification can be used to verify workflows as well. However, the same prerequisite applies as for knowledge base formalization and service/workflow description: Verification of service compositions – and their workflows – must rely on the same semantics as knowledge modeling, services, and workflows.

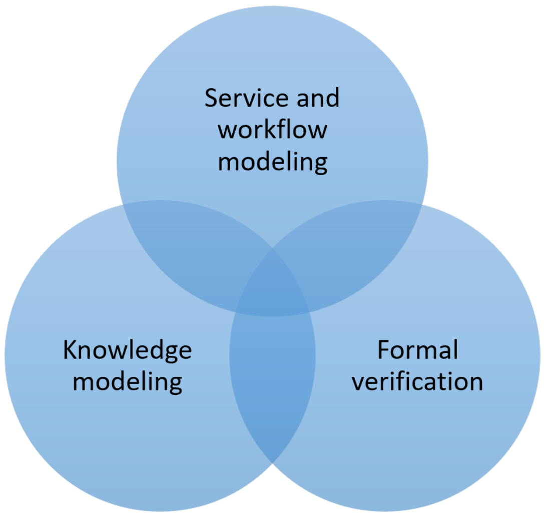

FIGURE 1.1: Joint research: A shared foundation of knowledge modeling, service and workflow modeling, and formal verification

To aim at the creation of *correct* service compositions as in the introductory OTF scenario, we thus identified three core topics, which are closely related: Knowledge base modeling, service description and composition, and formal verification. To make the result of a verification (and a composition) reliable in an OTF context, all of them must be based on the same semantics. This thesis solves this issue by providing a consequent proof of concept to show an integrated formalism to address all three topics. With *logical structures* and a logic-based semantics as a foundation for knowledge modeling, we provide a basis to create domain specific vocabulary. The vocabulary is used to describe services, with semantics also based on logic. The same is true for workflow semantics of service compositions. Based on the workflow semantics and a requirements specification, we define *correctness* of a service composition. Additionally, we define a Hoare-style proof calculus and relate its syntax-based proof outlines with the semantic-based correctness definition. While logic-based program verification as well as knowledge-based service modeling are active research topics, this thesis combines all three areas ($\rightarrow$ Figure 1.1).

The first core contribution provides a theoretical framework. As a complement, the second one aims at leveraging a practical implementation of the approach.

1.2 A Practical Implementation

Verification in the framework of this thesis is not necessarily automated. However, in an on-the-fly scenario, the overall process of creating and delivering a correct service composition has to be, by premise, “fast”. This context requires verification to be at least automated and at best fast and efficient. In practice, verification problems are not necessarily decidable, let alone efficient. We provide two steps to leverage practical verification: On a conceptual level, we enable verification of composition *templates* and the creation of service compositions which are *correct by construction*. On a pragmatic level, we encode the question whether a composition is correct as a satisfiability problem of a predicate logic formula. This is the second core contribution of this thesis.

In practice, an OTF provider cannot generate service compositions with arbitrary structures. However, if the general structure of a workflow is known, a workflow *template* can be modeled manually. This may be the case if (a) a task is generic with no regard to the business domain, or (b) the task is complex, but very specific to the domain and well understood by domain experts, though the exact implementation depends on actual requirements.

Case (a) includes all-purpose algorithms like sorting, filtering, or learning. Here, the actual “knowledge” is reduced to generic data, sets, and basic operations. This thesis uses a filter workflow as a running example in the following chapters. Case (b) includes complex tasks like the combination of domain specific classes of services without the need of selecting a specific service immediately. An example is the processing of data in several steps: Depending on the exact type of input data, different preprocessing and processing services are needed, though the overall structure will be the same. As an example, the input data has to be converted (using a conversion service) to the input of an appropriate analysis service. In both cases, a template provides the general structure of a workflow, using “service placeholders” instead of, or in addition to, actual services. The composition process then replaces the placeholders with services from an actual business domain.

A composition template can be verified without the timing constraints of an on-the-fly context. Instead, semi-automated or manual proofs can be used to show correctness. However, as a template does not use actual services, a proof of correctness has to include all possible instantiations, that is, compositions that can be generated from this template. Obviously, this is not possible, as this would include a correct proof for two alternative services, one having the opposite behavior of the other. Instead, we introduce *constraint rules* for a template. These constraints summarize the knowledge that is necessary to prove the template to be correct with respect to its pre- and postcondition. As they may contain references to service placeholders, the problem of two antagonistic services for one placeholder can be solved by selecting appropriate constraint rules. Therefore, a composition template is correct not only with respect to its pre- and postconditions, but also to its constraint rules. When it gets instantiated as a service composition by replacing service placeholders with actual services, it is sufficient to check whether the constraint rules can be concluded from the knowledge base of the target domain and the actual services used in the composition.

Constraint rules have the same structure as rules in a knowledge base. Checking for conclusion can therefore be handled within the domain of knowledge base reasoning, as the structure of the workflow does not have any direct impact. Chapter 4 formalizes the correspondence between constraint rules and correctness of the instantiation result as a theorem.

This first step towards automation replaces the need for verification of software compositions with a check for easier side conditions, whenever a composition is created by instantiating a correct template. The second step aims at verifying a template automatically. To this end, we utilize the consequent use of logic throughout our approach and formalize a theorem to relate provability of correctness of a template (with respect to its pre- and postcondition) with a logical satisfiability problem. To do this, we translate the workflow of the composition template and its pre- and postcondition into a logical formula. We then show that this formula is a tautology if and only if there exists a proof of correctness using the proof calculus for workflow templates. We also show that this relation is sound and complete. Chapter 5 elaborates on this encoding. Additionally, Chapter 7 discusses a prototypical translation into SMT-LIB, a standard solver input language (Barrett et al., 2015), along with an implementation.

The complete approach of service composition and verification relies on *descriptions*, that is, *models* of services. Therefore, proving correctness of a composition can only be treated as correctness of the *model* of the composition: It depends on whether or not the actual *behavior* matches the service description. An on-the-fly context provides two approaches to deal with that issue. From a verification point of view, a single service can be verified with respect to its description. If the description is true to the behavior, techniques like *proof carrying code* (Necula and Lee, 1998) or the generation of *certificates* (Jakobs and Wehrheim, 2014) can be used to provide an easily reproducible proof for the service, and its model can be marked as trustworthy. In that case, a model indeed reflects the behavior of the service. If this is not possible, economic techniques can be used to accumulate user feedback about earlier uses of the service, creating a *trust vector* or *reputation* (Mármol and Kuhnen, 2015). While this reputation is not the same as a measure for correctness, it can serve as a base for adding a probability value to correctness results: Services with a good reputation are treated as “description matches behavior”, while lower reputation values translate into a probability of failure – the service description may contain errors. While this is not the main part of this thesis, Chapter 6 introduces first ideas to join logic based verification with probabilities coming from reputation values.

1.3 Structure of this Thesis

This thesis presents three core contributions:

(1) A shared mathematical foundation for a verification of service compositions based on a formal knowledge base;

(2) A framework to reduce verification of service compositions to a check of easier side conditions, if the composition is created (instantiated) based on a correct composition template;

(3) A correspondence of correctness of a template with a satisfiability problem.

Chapter 2 (Domain Knowledge and Logic) discusses formal knowledge modeling in terms of *ontologies* and gives a definition of a knowledge base as used in this thesis. It also provides the necessary basic definitions of logic in general.

Chapter 3 (Workflow Descriptions) relates to current approaches of service modeling and workflows and gives formal definitions for both. For workflows, it defines an operational semantics and a notion of correctness. To prove or disprove correctness, a Hoare-style proof calculus is defined and proven to be sound and complete.

These chapters are the foundation for contribution (1), and Chapter 4 (Workflow Templates) introduces contribution (2). Based on workflows, it introduces workflow templates, a semantics, and a corresponding proof calculus. Core of this chapter is Theorem 4.24 (Constraint Rule Compliance): It reduces the need for composition verification to the check of constraint rules of a template.

Chapter 5 (Automating Correctness Proofs using First-order Logic) presents contribution (3) and defines a logical encoding of workflows. With Theorem 5.10 (Provability corresponds with Tautology), it relates correctness of a template (or composition) with the tautology of a logical formula, leveraging the automation of the verification process.

Chapter 6 (Dealing with Uncertain Service Descriptions) takes a detour and describes first ideas to integrate uncertainty of service descriptions into logic-based verification.

The pragmatic aspect of the automation of verification is covered by Chapter 7 (Prototypical Implementation). It documents an implementation of the logical encoding of Chapter 5.

Finally, the Conclusion summarizes the contributions of this thesis, and the Appendix provides additional example templates and compositions.

Part I

Preliminaries

Chapter 2

Domain Knowledge and Logic

Chapter 1 introduced the setting of on-the-fly service composition and verification in general. It presented three different research topics, namely (1) formal knowledge modeling, (2) service description and composition, and (3) program verification.

This chapter discusses the formal foundation for topic (1), *knowledge modeling* ($\rightarrow$ Figure 2.1). As logical structures and predicate logic in general are at the heart of a formal semantics of our definition of knowledge modeling, it consequently also addresses the first core contribution of this thesis, a mathematical foundation for a verification of service compositions based on a formal knowledge base. To this end, it discusses knowledge formalization, ontologies in general, and the use of *description logics*. Description logics are a subset of first-order logic and are used to model *ontologies*, and they provide the template to define the semantics of the *Web Ontology Language* (OWL). Combined with an additional *rules* layer, description logics are a versatile tool for knowledge modeling.

Based on these premises, we define our custom knowledge base for this thesis.

2.1 Formalizing Domain Knowledge

Discussing the formalization of pre- and postconditions of services includes discussing propositional and predicate logic. Within a fixed context, propositions and predicates cannot be chosen arbitrarily, but from the point of view of *knowledge formalization*, as service descriptions have to be backed by domain knowledge.

Knowledge formalization has its roots in ancient greek philosophy, which is also the source of the term “ontology”. Aristotle, in his works collectively known as *Organon* (Holzinger, 2013), introduced a concept-based logic and coined the term *Ontology* to specify the description, and its meaning, of the world. His works became the foundation for Descartes and Leibnitz, and later Boole, Frege, and Peano, leading to a theory of predicate logic (Davis et al., 1993, p. 22).

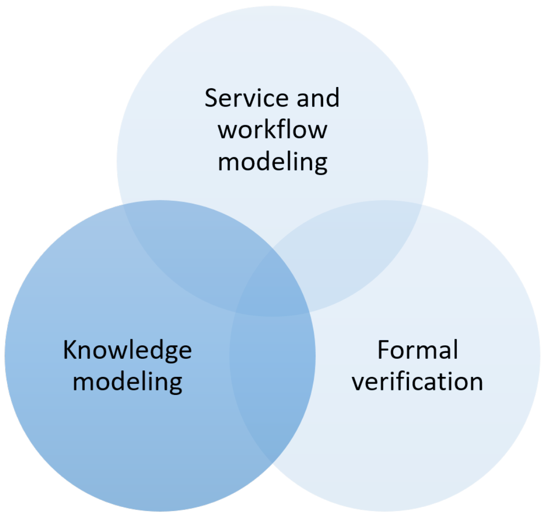

Figure 2.1: Joint research: Knowledge modeling in relation to service and workflow modeling and formal verification

In computer science, research about artificial intelligence (AI) is a main factor of pushing logical research further and further. Davis et al. highlight from an AI perspective which *roles* a knowledge representation has to fulfill. They also state that knowledge representation in AI is not restricted to logic, but also includes frames and semantic nets as well as rule-based approaches. Consciously choosing a style of representation is an important part of knowledge modeling, as it affects the way of *thinking* about knowledge, and reasoning about this knowledge. For example, in logic, the focus is on single individuals and their relationship, while with frames we think about prototypical objects, descriptions, or situations. Contrasting this, rules support an empirical, evidence-based approach (Davis et al., 1993, p. 20).

By choosing a formal representation, we do not only make a premise about the *subjects* of our reasoning, but also about the overall *idea* of what we consider as meaningful reasoning. If we, e.g., opt for a *logical* representation with sound inference rules, we argue that intelligent reasoning *itself* actually *is* sound, logical reasoning, and vice versa: Only logical, sound reasoning is intelligent and useful. While this is indeed an appropriate claim in the context of formal, precise service descriptions, this is in no way the only possible option to formalize knowledge, and to reason about knowledge. Unsound reasoning has its applications as well (e.g., as *inductive reasoning* to generate hypotheses for observed data) and the same is true for completely different approaches, where knowledge and reasoning is modeled after human experts, involving concepts like goals and plans. In this scenario, stemming from psychology, “intelligent” reasoning over a formalized knowledge is treated as a complex, not necessarily completely analyzable, property of human behavior (Davis et al., 1993, p. 23).

While it is useful to keep these inherent modeling decisions in mind, opting for a logical knowledge representation comes quite naturally in our scenario, even from this broader perspective: Service descriptions and requirements have to be described as precise and analyzable as possible, with results which transfer between knowledge representation, service modeling, and verification. On a more technical level, treating data (or variables holding these data) as “individuals” also seems reasonable, and a formalization using logic is an appropriate consequence. With this in mind, we choose *ontologies* (in a computer science sense) to model knowledge. While ontologies can be (and are) used to model *taxonomies*, or tree-shaped classification graphs, their capabilities include more complex relationships than mere tree-like dependencies between predicates, especially with additional rules languages. Section 2.2 provides more details.

On a formal level, there are different approaches to model knowledge in terms of ontologies. However, they share their most basic property: In the end, they define a *vocabulary* (concepts and their relations), which have to have an evaluation to decide the question “Is some expression, written in terms of the ontology’s vocabulary, *true* or *false*?” This evaluation is done by finding a logical *interpretation* which respects the knowledge modeled in an ontology. At the core of this process are *logical structures*, which do not only include the *domain* of all variables (typically, the Boolean domain, but especially if we talk about predicates, variables will have additional domains, too), but interpretations of predicates. Because logical structures are the most basic part of logic, and because they play a vital role in the latter parts of this thesis, we define them formally.

Logical structures assign an interpretation to variables and constants. Both variables and constants have an associated type, which defines the available set of values that a variable or constant can have. This set of values for a type is defined by the *universe* or *domain* of the type, which is part of the logical structure. While the values of variables can change, the values of constants are fixed. For now, we assume that variables and constants are sets of unique names.

Definition 2.1 (Variables and Constants)**.** Let *Var* be a set of unique names which serves as variable names, and let *Const* be a set of constant names.

Later in this thesis, we define them with a type association ($\rightarrow$ Definition 2.7). Apart from that, definitions can be found in any textbook about logic; here, we follow Apt, de Boer, and Olderog (2009, p. 31-33), as we do later as well.

Definition 2.2 (Logical Structure)**.** We call $\mathcal{S} = (\mathcal{U}, \mathcal{I})$ a *logical structure* with a universe (or domain) $\mathcal{U}$ and interpretation $\mathcal{I}$, which maps constants c to their value $\mathcal{I}(c) \in \mathcal{U}$.

We extend the notion of interpretations when we introduce types and terms.

2.2 Description Logics, Ontologies, and Knowledge Bases

While the last section motivated knowledge formalization in general, this section introduces ontologies formally, their representation using *description logics*, and extending frameworks for *rules* to enhance their expressiveness. After that, we formalize a knowledge base in a way that is used throughout the remainder of this thesis.

2.2.1 Description Logics as Ontology Semantics

Ontologies are a way to formalize knowledge, with different formalizations available. Gruber formulated the idea of making knowledge "portable", to share it among different AI systems. He gives the archetypical characterization of ontologies (for computer scientists) in Gruber (1993, p. 199):

> An *ontology* is an explicit specification of a conceptualization.

Basically, every terminological description of the world, or a domain, introduces a *conceptualization*. An ontology is a formal specification of the set of concepts, roles relating these concepts, and individuals which may belong to one or more concepts. An important difference is the distinction between *explicit* and *implicit* conceptualizations, one stating properties of single individuals, while the other characterizes groups of individuals. Explicit and implicit conceptualizations follow two different approaches of describing knowledge of a world.

Explicit conceptualization is a direct approach. Here, knowledge of every individual of a world is formalized directly, that is, individuals are uniquely identified (enumerated), and predicates apply directly to them. As a description of knowledge of a world consists of a collection of facts about individuals, the complete description changes as soon as a fact about a person changes. Imagine a person identified as "Bob" is a "team leader" at first (denoted as *isTeamLeader(Bob)*). As a team leader he has to respond to "Charlie", who is head of department (*isHeadOfDepartment(Charlie)* and *respondsTo(Bob, Charlie)*). Later, he becomes head of department himself (*isHeadOfDepartment(Bob)*). This causes additional changes in the set of facts which describe the world, as he now does no longer respond to Charlie, but to Alice, who is CEO (*respondsTo(Bob, Alice)*). The reason is that some predicates (to whom someone has to respond) are modeled explicitly for all individuals, but they actually depend on the position of an individual in the company, not the individual itself.

As this is cumbersome for reasoning about knowledge, implicit conceptualizations aim for a separation of concerns between stating facts about *terminology*, and *individuals*. In the example, *respondsTo* would relate positions (head of department and CEO) rather then individuals, so when Bob gets promoted, only his positional predicate is changed. This way, implicit conceptualizations provide a vocabulary of a domain which can be used independently of individuals. Guarino, Oberle, and Staab (2009) provide an excellent step-by-step formalization from explicit to implicit conceptualizations including the impact on both expressiveness and usability.

To specify ontologies in general it is important to understand these differences between grades of granularity of an ontological representation. However, this does not automatically decide about a formal *representation*, or, the ontology *language*. Guarino et al. use an ontology language only as a parameter in their formalizations, as different needs

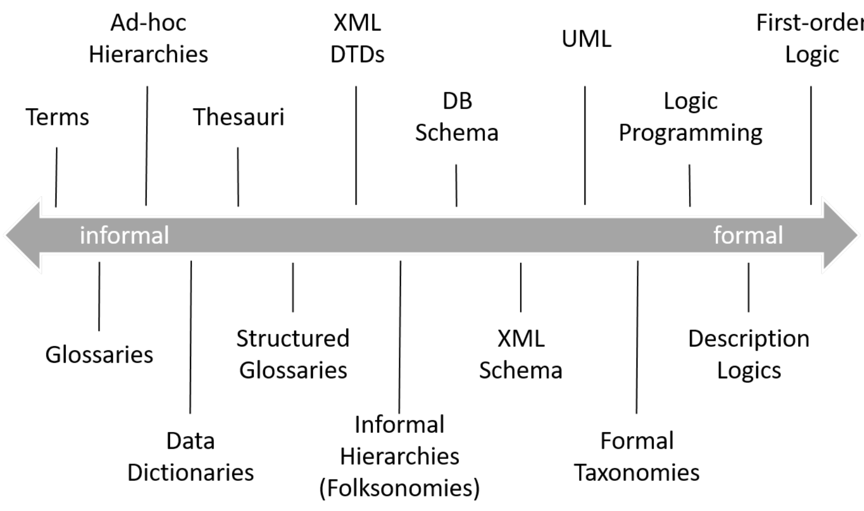

FIGURE 2.2: Range of ontology formalization, excerpt from Guarino et al. (2009, p. 13)

call for different languages. These needs range from very informal to very formal languages (→ Figure 2.2). A collection of terms, or an ordinary glossary (as in textbooks), mark the *informal* end of this range, and logical languages, like functional programming languages, description logics, and predicate logic, mark the *formal* end (Guarino et al., 2009, p. 12-14). As our goal is formal specification and reasoning, we need a *formal* representation language, more specifically, predicate logic.

For ontologies, logical languages are already common as representation languages, because formal reasoning is often a goal in knowledge representation. Other goals are *decidability* of reasoning problems in general and *efficiency* of a decision procedure more specifically. This leads to languages with restricted expressiveness, but good computability, and *description logics* are a modular approach to customize a logic for both of them. Starting with *attributive concept descriptions with complements* ($\mathcal{ALC}$), and the paper with the same title from Schmidt-Schauß and Smolka (1991), *description logics* (DLs) provide a modular family of logical languages which are inductively built from *concepts* and *roles*, which relate concepts. Description logics are *modular* in the sense, that additional classes of expressiveness can be added explicitly. The "modules" which are part of a specific description logic are typically denoted by letters. Based on a set of concept names and a set of role names, new concepts are created based on *concept constructors.* These constructors are related to Boolean connectives in propositional or predicate logic. The following example denotes the concept of a "parent" based on the concept "human" and the role "hasChild":

$$\textsc{Human} \sqcap \exists \textsc{hasChild}.\textsc{Human}$$

Description logics can be translated to predicate logic, using concepts as unary predicates and roles as binary predicates. Translation of concept constructors is straightforward.

Table 2.1: Description logic expressiveness abbreviations

Logic	**Properties**
$\mathcal{H}$	role hierarchies/subroles
$\mathcal{O}$	nominals/named individuals
$\mathcal{I}$	inverse roles
$\mathcal{N}$	number restrictions
$\mathcal{Q}$	qualified number restrictions
(D)	concrete domain, e.g., reals, integers
$\mathcal{R}$	complex role inclusion

In predicate logic, the example is:

$$\forall h : \textsc{Human}(h) \land \exists i : \textsc{hasChild}(h, i) \land \textsc{Human}(i)$$

For a complete definition of syntax and semantics of description logics, we refer to Schmidt-Schauß and Smolka (1991) and Baader et al. (2003, 2008), Guarino et al. (2009).

More complex description logics are created by adding more expressiveness. The most common extension is to allow for *transitive roles*, leading to the logic $\mathcal{ALC}_{R+}$, abbreviated as $\mathcal{S}$ due to its similarity to the modal logic S4 (Baader et al., 2008). Other properties include the use of named individuals, role hierarchies, number restrictions, or the use of concrete domains like integers. Table 2.1 lists common extensions.

2.2.2 Reasoning in DLs: General Concept Inclusion

It is possible to create non-trivial concepts where the interpretation is empty, or, in a predicate logic terminology, which are contradictory, e.g., $\textsc{Room} \sqcap \neg\textsc{Room}$. Schmidt-Schauß and Smolka define a concept C as *coherent*, if its interpretation is non-empty: therefore, $\textsc{Room} \sqcap \neg\textsc{Room}$ is incoherent (Schmidt-Schauß and Smolka, 1991, p. 5). Coherence can be used to express *subsumption*. A concept description C is *subsumed* by a concept description D (denoted as $C \sqsubseteq D$), if the logical interpretation is a subset of the interpretation of D, which is equivalent to the incoherence of $C \sqcap \neg D$ (Schmidt-Schauß and Smolka, 1991, p. 5). More intuitively, $\textsc{House} \sqsubseteq \textsc{Building}$ is a subsumption in the domain of architectural structures, as every house is indeed a building. The similarity of the subsumption symbol $\sqsubseteq$ to the subset relation symbol $\subseteq$ is intentional. Subsumption, or *general concept inclusion* (GCI), is the basic tool to restrict knowledge within a domain.

As GCIs work solely on the vocabulary of a domain knowledge, that is, concepts of an ontology, their restrictions work on a *terminological* level. In ontology reasoning, a set of GCIs is called a *TBox*. In description logics which include role hierarchies ($\mathcal{H}$), the

TBox does not only include general concept inclusions, but also *role inclusions* to model these role hierarchies. In contrast to TBoxes, an *assertional* level of restricting domain knowledge is called an *ABox* (e.g., Baader et al., 2009). A TBox restricts relations between concepts directly, while an ABox reasons about *individuals* (which is necessary, e.g., in description logics with the $\mathcal{O}$ expressiveness). As the separation into TBox and ABox is mainly a design choice to classify reasoning problems in description logics, we will not use the terms in this thesis. We refer to, e.g., Baader et al. for a more detailed introduction to semantics, exemplified on the DL $\mathcal{SHIQ}$.

To check whether a logical structure is a model of a set of GCIs, the idea of *coherence* from above is used. As some logical structure $\mathcal{S}$ is a model of a general concept inclusion $C \sqsubseteq D$ if and only if the concept $C \sqcap \neg D$ is coherent, subsumption checking amounts to coherence checking, which can be modeled as a satisfiability problem. Depending on the actual description logic, this satisfiability problem is decidable, and several specialized ontology reasoners exists to decide coherence problems, e.g., HermiT[1] (Glimm et al., 2014), Pellet[2] (Sirin et al., 2007), and FaCT++[3] (Tsarkov and Horrocks, 2006).

2.2.3 Higher-level Ontology Languages

With the advent of the *Semantic Web* (Berners-Lee et al., 2001) it became necessary to formalize ontologies in a way which was closer to the notations common in software engineering disciplines. The *resource description framework* (RDF) provides an XML-based, structured way to describe relations between concepts (Gandon and Schreiber, 2014). Though initially created as a standard to describe resources (e.g, uniform resource identifiers/URIs), today it serves as a fundamental base for ontology languages. RDF statements are triples, which relate two concepts, or *RDF types*, with an *RDF predicate.* RDF types are custom names (just as concepts), and while predicate names are also custom, they require two types which they link.

RDF triples are the backbone of the *Web Ontology Language* (OWL, Patel-Schneider et al., 2012). The Web Ontology Language, a W3C[4] recommendation, provides different syntaxes to formalize knowledge, mainly by introducing syntax elements to conveniently formulate complex knowledge expressions. The foremost syntax is an RDF-based XML syntax, which serves as a standard for tool interoperability (Gandon and Schreiber, 2014). Other syntaxes, like *Manchester syntax* (Horridge and Patel-Schneider, 2012), provide a more concise notation similar to logic.

As reasoning, and therefore decidability, is always an issue in knowledge representation, OWL defines two (in earlier versions: three) subsets of language constructs, to cater different needs of expressiveness, and therefore decidability: *OWL Full* provides the complete OWL syntax. Its semantics is based on the semantics of RDF (Hayes and

[1]HermiT, `http://www.hermit-reasoner.com`, retrieved on July 21, 2017.
[2]Pellet, `https://github.com/stardog-union/pellet`, retrieved on July 21, 2017.
[3]FaCT++, `https://bitbucket.org/dtsarkov/factplusplus`, retrieved on July 21, 2017.
[4]World Wide Web Consortium, `https://www.w3.org`, retrieved July 21, 2017.

Patel-Schneider, 2014, Schneider, 2012). *OWL DL* is a syntactical subset of OWL Full, and while it can also be interpreted in RDF semantics, it comes with a description logic style semantics, or *direct semantics* (Grau et al., 2012). The direct semantics is closely related to the description logic $\mathcal{SROIQ}$, but to ensure decidability while making use of features like transitive roles, it comes with additional restrictions, for example, quantified number restrictions ($\mathcal{Q}$) are not allowed for transitive roles (Patel-Schneider et al., 2012, Section 11). The expressiveness of OWL Full goes beyond DL compatibility. An example is the treatment of OWL classes both as (terminological) classes and as individuals at the same time. Reasoning over OWL Full is undecidable.

OWL (version 2) is a W3C recommendation, and supported by various modeling and reasoning tools, like Protégé[5] (Horridge et al., 2014, Knublauch et al., 2005), for modeling, and by the reasoners, e.g., Pellet, HermiT, and FaCT++. This and the incorporation of XSD standard types like strings, integers, and floating point decimals, lead to OWL being a common knowledge modeling language in the context of the Semantic Web (W3C, a,b).

2.2.4 Enhancing Ontologies with Rules

Using ontologies with well researched languages like DLs it is possible to express complex relationships. This is especially true for more expressive DLs with complex role properties, which leave the realm of decidability. Because "undecidable in general" does not necessarily mean "undecidable in practice", it is possible to extend the expressiveness of ontologies even further. Depending on the exact knowledge to be modeled, this is even necessary. A typical relationship which cannot be expressed with a description logic like $\mathcal{ALC}$ is the "uncle problem" (e.g., Parsia et al. 2005, p. 12). Consider the following roles (or binary predicates) in a DL: $\textsc{parentOf}, \textsc{siblingOf}$, $\textsc{uncleOf}$. We try to state the following equivalence: If my sibling has a child, I am the uncle. It is not possible to state this kind of triangular relationship in $\mathcal{ALC}$. However, it is no problem at all to state this in predicate logic:

$$\begin{aligned}\forall \mathit{person}, \mathit{sibling}, \mathit{child} : \quad & \textsc{siblingOf}(\mathit{person}, \mathit{sibling}) \wedge \textsc{parentOf}(\mathit{sibling}, \mathit{child}) \\ & \Rightarrow \textsc{uncleOf}(\mathit{person}, \mathit{child})\end{aligned}$$

The need to express such properties leads to the addition of *rules* to DL ontologies. Generally, rules take the form of logical implications, using predicates (or expressions based on predicates) as antecedent and consequent. Adding rules does not make the subsumption problem undecidable by itself – it depends of the exact style of rules. Some rules can be expressed using GCIs quite easily and therefore are a form of "syntactic sugar". Stating that all humans are mammals, using

$$\forall x : \textsc{Human}(x) \Rightarrow \textsc{Mammal}(x)$$

[5]Protégé, `http://protege.stanford.edu`, retrieved on July 21, 2017.

can be expressed as a general concept inclusion as well:

$$\textsc{Human} \sqsubseteq \textsc{Mammal} .$$

There are several approaches to combine a structural knowledge base with rules. Levy and Rousset combine description logics with Horn rules (Levy and Rousset, 1996). Donini, Lenzerini, Nardi, and Schaerf use $\mathcal{ALC}$ to model the structural part of a knowledge base and Datalog (Gallaire and Minker, 1978, Ullman, 1988) to model a relational part, resulting in $\mathcal{AL}$-log (Donini et al., 1998). They also limit themselves to Horn clauses, or *positive* Datalog. Additionally, they require that variables in the head of a rule (the consequent) also appear in the body of the rule (the antecedent). This way, they keep reasoning decidable. Rosati extends their approach to *disjunctive* Datalog (Eiter et al., 1997), therefore allowing for negation and disjuncts in the body of a rule (Rosati, 1999). Motik, Sattler, and Studer do not restrict themselves to $\mathcal{ALC}$, but build on OWL DL (as they work on OWL version 1, that is basically $\mathcal{SHOIN}$, in contrast to the current OWL2 $\mathcal{SROIQ}$; Motik et al., 2005). While their description logic is more expressive than $\mathcal{ALC}$, and therefore a more complex structural knowledge base is possible, they also restrict themselves to Horn clauses, or positive Datalog, to keep decidability.

Ultimately, this leads to the semantic web rule language (SWRL), which tightly integrates with OWL (Motik et al., 2009). Though SWRL is not a W3C standard yet (but a submission[6]), its tight integration with the OWL modeling tool Protégé leverages its use in rule-supported ontology modeling. SWRL itself is a subset of Datalog, and more expressive than OWL DL. In an OWL context, it is restricted by additional constraints to keep it decidable (Parsia et al., 2005).

Typically, rules are added to a knowledge base language very conservatively, as decidability is a major concern. Rosati discusses the complexity of the combination of ontologies and rules more thoroughly (Rosati, 2005). To ease the comparison between different approaches to rule languages, Franconi and Tessaris propose a framework to formalize rules in a general way, including Datalog-based approaches (Franconi and Tessaris, 2004).

Summarizing, decidability of a rule-enhanced ontology results from the underlying description language as well as the structure of the rules. For the context of this thesis, we do not restrict the structure of rules, and therefore expressiveness as well as decidability, in general. Instead, decidability will depend on the concrete domain knowledge, and the rules necessary to model it, which are part of a concrete use of the approach of this thesis.

[6]`https://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`, retrieved July 21, 2017.

Table 2.2: DL concept constructors as rules, simplified representation of the construction rules of (Baader et al., 2008, p. 144)

DL	**Rule**
$\neg C$	$\forall x : \neg C(x)$
$C \sqcap D$	$\forall x : C(x) \wedge D(x)$
$C \sqcup D$	$\forall x : C(x) \vee D(x)$
$\exists r.D$	$\forall x \exists y : r(x,y) \wedge D(y)$
$\forall r.D$	$\forall x,y : r(x,y) \Rightarrow D(y)$

2.2.5 A Formal Knowledge Base

Up to now, we introduced the basic formalisms to model domain knowledge using ontologies, with a logic-based semantics. Depending on the exact domain knowledge, the modularity of description logics is an important tool to find the exact level of expressiveness which is needed, aiming at decidability of the subsumption problem.

As the goal of this thesis is the union of knowledge modeling, service description, and program verification, the exact expressiveness of a concrete description logic is not as important as it may seem. On the contrary, we will define a knowledge base for domain knowledge in a generic way, without providing an explicit set of different modules with different expressiveness as in description logics, and allow for a large expressiveness. This way, decidability, and complexity of the resulting satisfiability problem depends on the concrete domain knowledge which is formalized, and therefore may vary between domains. If, e.g., qualified number restrictions (domain logic $\mathcal{Q}$) or cyclic rules are not necessary to model knowledge of a domain (which is the case for simple taxonomies), then this is beneficial for decidability.

We roughly follow the notation introduced with description logics, but with some important changes: We use the GCI symbol $\sqsubseteq$ to model concept inclusions (that is, *concept hierarchies*), directly as part of the knowledge base, and we use *rules* to model any other additional constraints. This way, we are able to define a knowledge base quite easily. As another syntactical difference to description logics, we do not use concept constructors and role quantifiers; see Table 2.2 for their relation to predicate logic. The translation follows (Baader et al., 2008, p. 144). Instead, we directly define *concept and role predicates*, based on concept and role names, to create logical expressions.

Definition 2.3 (Knowledge Base)**.** Let $\mathbf{I}$ be a set of individuals. A *rule-enhanced knowledge base* (or just knowledge base) K consists of a tupel $K = (C, P, \sqsubseteq, R)$, with:

- C a set of (unique) concept names, with $A(x)$ a concept predicate for $A \in C$ and $x \in \mathbf{I}$,
- P a set of (unique) role (or predicate) symbols, where every $p \in P$ is associated with a relation $p \in C \times C$, and $p(x,y)$ a role predicate for $p \in P$ and $x, y \in \mathbf{I}$,

- $\sqsubseteq$ a partially ordered subconcept relation $\sqsubseteq \subseteq C \times C$,
- R a set of constraint rules.

A constraint rule $r \in R$ is a boolean expressions constructed as:

$$\begin{aligned} A(x) &\in R \\ p(x, y) &\in R \\ X \in R \;\Rightarrow\; \neg X &\in R \\ X, Y \in R \;\Rightarrow\; X \vee Y &\in R \;. \end{aligned}$$

In description logics, role names are just names and used by concept constructors to define new, composed concepts, which use can be expressed in predicate logic as binary predicate. In practice, this is used to define domain and range of a role, e.g., to restrict the role "married with" to let only "humans" be married, we write $\exists$MARRIEDWITH $\sqsubseteq$ HUMAN, that is, every concept which has a role "married with" has to be subsumed by the concept "Human". If we do the same for the inverse role, we restrict both domain and range to humans. In the definition above, we included typing of role predicates directly, to simplify the notation towards usual functional notation, that is, we write MARRIEDWITH : HUMAN $\times$ HUMAN to express the same restriction as in the example.

We define the semantics of a knowledge base using logical structures ($\rightarrow$ Definition 2.2).

Definition 2.4 (Semantics of a Knowledge Base)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base. Let *Var* be a set of variables and *Const* a set of constants, which serve as set of individuals $\mathbf{I} = \mathit{Var} \cup \mathit{Const}$. Let $\mathcal{S} = (\mathcal{U}, \mathcal{I})$ be a logical structure with an interpretion $\mathcal{I}$ and a universe $\mathcal{U}$, where $\mathcal{U} = \mathcal{U}_{C_1} \cup \cdots \cup \mathcal{U}_{C_n}$ for every $C_1, \ldots, C_n \in C$. We define the *semantics* of K by $\mathcal{S}$, where

$$\begin{aligned} &\mathcal{I} : C_1 \rightarrow 2^{\mathcal{U}_{C_1}} && \text{with } C_1 \in C \\ &\mathcal{I} : p_1 \rightarrow 2^{\mathcal{U}_{C_1} \times \mathcal{U}_{C_2}} && \text{with } p_1 \in P \text{ and } p_1 : C_1 \times C_2 \;. \end{aligned}$$

$\mathcal{S}$ satisfies K, that is, $\mathcal{S} \models K$, if and only if it satisfies all subconcept relations $A \sqsubseteq B$ and all constraint rules $r \in R$. For $A, B \in C$ and $p \in P$, therefore:

$$\mathcal{S} \models (A \sqsubseteq B) \;\Leftrightarrow \mathcal{I}(A) \subseteq \mathcal{I}(B)$$

$$\mathcal{S} \models r \;\Leftrightarrow\; \begin{cases} r \equiv A(a) : & \forall a \in \mathbf{I} : \mathcal{I}(a) \in \mathcal{I}(A) \\ r \equiv p(a, b) : & \forall a, b \in \mathbf{I} : \big(\mathcal{I}(a), \mathcal{I}(b)\big) \in \mathcal{I}(p) \\ r \equiv \neg q : & \mathcal{S} \not\models q \\ r \equiv q \vee t : & \mathcal{S} \models q \text{ or } \mathcal{S} \models t \end{cases}$$

As $\sqsubseteq$ is reflexive (because it is a partial order), every concept is a subconcept of itself, which matches the notion of GCIs. We use the term *rules* to denote the constraint rules

R to follow the convention of ontology formalization, even if they are not necessarily implications.

Example 2.1 (Tourism Domain: Restaurants)**.** *The ontology $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ formalizes a small excerpt from a domain of Tourism concepts, which we will use throughout this thesis. We will use the subscript D for "domain-specific ontology". The part formalized in this thesis deals with the relation of* restaurants, ratings*, and* price *values assigned to restaurants, and the predicates relating these concepts. Here, a* Snack Bar *is a special kind of* Restaurant *to demonstrate subtyping.*

$$\begin{aligned}
C_D = \{&\textsc{Site}, \textsc{Restaurant}, \textsc{SnackBar},\\
&\textsc{InfoTag}, \textsc{Rating}, \textsc{Michelin}, \textsc{Price}, \textsc{Location},\\
&\textsc{Distance}, \textsc{Event}, \textsc{Date}\}\\
P_D = \{&\textsc{hasRating} : \textsc{Restaurant} \times \textsc{Rating}, \textsc{isMinRating} : \textsc{Rating} \times \textsc{Bool},\\
&\textsc{isRatingLess} : \textsc{Rating} \times \textsc{Rating}\\
&\textsc{hasPrice} : \textsc{Restaurant} \times \textsc{Price}, \textsc{isMaxPrice} : \textsc{Price} \times \textsc{Bool},\\
&\textsc{goodRestaurant} : \textsc{Restaurant} \times \textsc{Bool},\\
&\textsc{isBetterRestaurant} : \textsc{Restaurant} \times \textsc{Restaurant},\\
&\textsc{cheap} : \textsc{Restaurant} \times \textsc{Bool},\\
&\textsc{hasLocation} : \textsc{Site} \times \textsc{Location},\\
&\textsc{distFrom} : \textsc{Distance} \times \textsc{Location}, \textsc{distTo} : \textsc{Distance} \times \textsc{Location},\\
&\textsc{distValue} : \textsc{Distance} \times \textsc{Integer},\\
&\textsc{hasEvent} : \textsc{Site} \times \textsc{Event}, \textsc{startsAt} : \textsc{Event} \times \textsc{Date}\}\\
\sqsubseteq_D = \{&(\textsc{Restaurant}, \textsc{Site}), (\textsc{SnackBar}, \textsc{Restaurant}),\\
&(\textsc{Michelin}, \textsc{Rating}), (\textsc{Rating}, \textsc{InfoTag})\}\\
R_D = \{&\textsc{hasRating}(\mathit{res}, \mathit{rat}) \wedge \textsc{hasRating}(\mathit{res2}, \mathit{rat2})\\
&\ \Rightarrow \textsc{isRatingLess}(\mathit{rat}, \mathit{rat2}) = \textsc{isBetterRestaurant}(\mathit{res2}, \mathit{res}),\\
&\textsc{hasRating}(\mathit{res}, \mathit{rat}) \wedge \textsc{isMinRating}(\mathit{rat}) \Rightarrow \textsc{goodRestaurant}(\mathit{res}),\\
&\textsc{hasRating}(\mathit{res}, \mathit{rat}) \wedge \neg\textsc{isMinRating}(\mathit{rat}) \Rightarrow \neg\textsc{goodRestaurant}(\mathit{res}),\\
&\textsc{hasPrice}(\mathit{res}, \mathit{price}) \wedge \textsc{isMaxPrice}(\mathit{price}) \Rightarrow \textsc{cheap}(\mathit{res}),\\
&\textsc{hasPrice}(\mathit{res}, \mathit{price}) \wedge \neg\textsc{isMaxPrice}(\mathit{price}) \Rightarrow \neg\textsc{cheap}(\mathit{res}),\\
&\mathit{trichotomous}(\textsc{isRatingLess}), \mathit{transitive}(\textsc{isRatingLess}),\\
&\mathit{irreflexive}(\textsc{isRatingLess}), \mathit{antisymmetric}(\textsc{isRatingLess}),\\
&\mathit{trichotomous}(\textsc{isBetterRestaurant}), \mathit{transitive}(\textsc{isBetterRestaurant}),\\
&\mathit{irreflexive}(\textsc{isBetterRestaurant}), \mathit{antisymmetric}(\textsc{isBetterRestaurant}),\\
&\mathit{functional}(\textsc{hasRating}), \mathit{functional}(\textsc{hasPrice}),\\
&\mathit{functional}(\textsc{isMinRating}), \mathit{functional}(\textsc{isMaxPrice}),\\
&\mathit{functional}(\textsc{goodRestaurant}), \mathit{functional}(\textsc{cheap}),\\
&\mathit{functional}(\textsc{hasLocation}), \mathit{functional}(\textsc{startsAt}),\\
&\mathit{functional}(\textsc{distFrom}), \mathit{functional}(\textsc{distTo}), \mathit{functional}(\textsc{distValue})\}
\end{aligned}$$

Figure 2.3 visualizes subset relations and roles of the knowledge base.

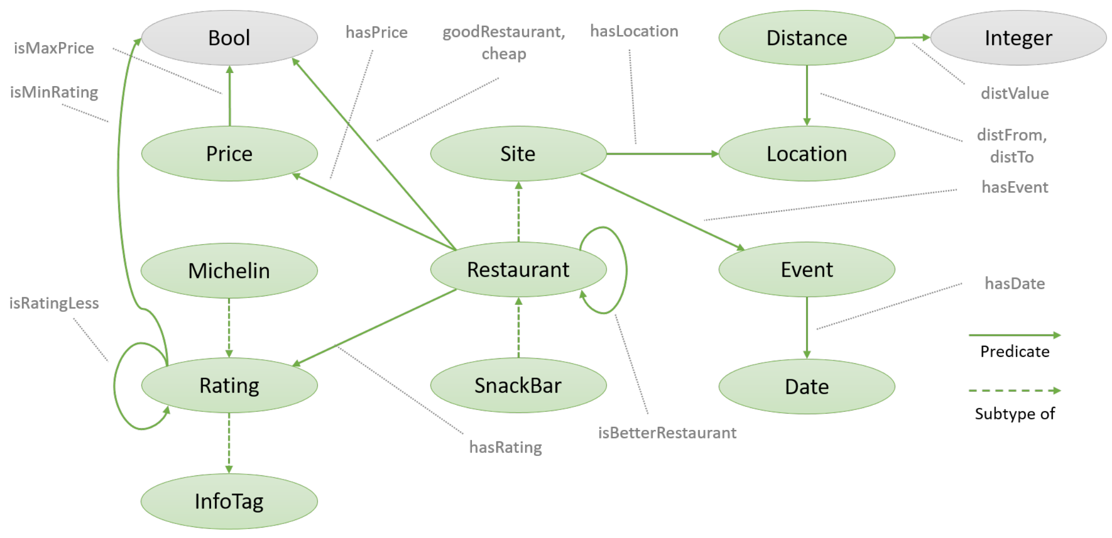

Figure 2.3: Visualization of the knowledge base of tourism (Example 2.1)

The rational behind the subconcept relation $\sqsubseteq$ is a polymorphic type system. Concepts and roles from knowledge modeling serve different purposes in service specification: While both represent the *vocabulary* used in service specifications, roles (binary predicates) are used in pre- and postconditions to describe characteristics of data, and concepts are used as *types*. There is no conceptual difference, as type information could also be encoded using unary predicates in pre- and postconditions and using untyped signatures for services at the same time. However, as service modeling stems from the domain of programming and modeling languages, the notion of typed parameters in method signatures is prevalent. Existing languages like OWL-S (Martin et al., 2005), which combine the domain of service/software modeling and ontologies, follow the approach of typed signatures combined with logical pre- and postconditions. Supporting the same idea, the subconcept relation paves the way for an elegant notation of concept hierarchies, which directly translate into a simple, hierarchical type system.

Before we define a type system formally, we borrow an additional simplification from OWL: OWL comes with XSD standard types. Theoretically, we can include any type as concept in a knowledge base. Practically, we assume that *Boolean* and *Integer* are always included in the set of concepts of a knowledge base. Additionally, we assume that all relations on Boolean and Integer are part of the roles, and their restrictions part of the rules. Obviously, this does only cover binary relations and does *not* cover functions. In DL-based ontologies, roles are always binary. However, it is possible to model n-ary roles as well by introducing an additional concept to represent the n-ary role, and introduce roles to connect it to its operands.

Example 2.2 (N-ary Roles)**.** *Let* $K = (C, P, \sqsubseteq, R)$ *be a knowledge base. Assume a ternary predicate, or role,* $hasDistance(l_1, l_2, d)$ *to model the* Distance d *between two* Locations l_1, l_2*, with*

$$\textsc{Location}, \textsc{Distance} \in C \ .$$

Then we introduce a concept $\textsc{HasDistance} \in C$ *and roles*

$$\textsc{distFrom}, \textsc{distTo}, \textsc{distValue} \in P$$

with

$$\begin{aligned} \textsc{distFrom} &: \textsc{HasDistance} \times \textsc{Location} \\ \textsc{distTo} &: \textsc{HasDistance} \times \textsc{Location} \\ \textsc{distValue} &: \textsc{HasDistance} \times \textsc{Distance} \ . \end{aligned}$$

We restrict the new roles appropriately, that is, in this case, they are functional.

Role properties like functionality can be expressed using the rules of a knowledge base. However, again we follow OWL notation and introduce notational shortcuts for typical properties of relations. Instead of writing, e.g.,

$$\forall x, y, z : \textsc{distValue}(x, y) \wedge \textsc{distValue}(x, z) \Rightarrow y = z \ ,$$

we just write

$$\textit{functional}(\textsc{distValue}) \ .$$

We do the same for other important properties like transitivity, reflexivity, and so on.

It is relatively easy to include standard types in a knowledge base, and this should be the same for set types, as long as we restrict ourselves to relations (we will discuss functions further below). However, the way how we will encode domain knowledge logically later in this thesis advocates to model sets *in addition* to a domain knowledge. Instead of modeling sets as default concepts of a knowledge base, we include them in the set of types which can be derived from an ontology. This definition is a main connecting element between ontology modeling and service modeling.

Definition 2.5 (Types)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base. We define the set of *types* $\mathcal{T}_K$ inductively as follows:

$$\begin{aligned} t \in C &\rightarrow t \in \mathcal{T}_K \\ T \in \mathcal{T}_K &\rightarrow \textbf{set}\ T \in \mathcal{T}_K \ , \end{aligned}$$

where **set** T is a set type whose elements are of type T.

With an explicit set of types $\mathcal{T}_K$ for a knowledge base K, we are able to cover set types. We include relations on types (e.g., $<$ on integers) implicitly in the roles of K. To cover relations on sets, we define the set of *predicates* derived from a knowledge base analogously to the set of types. Similarly, we define a set of *functions* on the standard types and sets.

Definition 2.6 (Predicates and Functions of a Knowledge Base)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base and $\mathcal{T}_K$ the set of derived types. Then $\mathcal{P}_K$ denotes the set of *predicates* and $\mathcal{F}_K$ the set of *functions* for K, with

- $P \subset \mathcal{P}_K$,
- common relations on Boolean and Integer $\subset \mathcal{P}_K$,
- common relations on set types $\subset \mathcal{P}_K$,
- common functions on Boolean and Integer $\subset \mathcal{F}_K$,
- common functions on set types $\subset \mathcal{F}_K$.

This definition includes that the set of functions depends only on the standard types Boolean and Integer, which we included in the domain knowledge for convenience.

Now, we *model* domain knowledge itself with a knowledge base, and are able to create *expressions* built on that knowledge using additional predicates and functions. While expressions are similar to concept constructors and role assertions in description logics, they are more expressive by including functions on standard types. As in the definition of the semantics of description logic and knowledge bases, *individuals* are used as operands of predicates. Here, we assume a set of variables *Var* to be used as individuals and extend Definition 2.1 with type information.

Definition 2.7 (Set of Variables and Constants)**.** Let *Var* be a set of unique variable names with an associated type, defined by $type : Var \to \mathcal{T}$. Let *Const* be a set of constant names with an associated type, defined by $type : Const \to \mathcal{T}$.

We define terms in the usual way.

Definition 2.8 (Terms)**.** Every $a \in Var$ is a term of type $type(a)$. Every $c \in Const$ is a term of type $type(c)$. If x, y are terms, then $p(x, y)$ with $p \in \mathcal{P}_K$ and $f(x, y)$ with $f \in \mathcal{F}_K$ are terms.

Please note that common Boolean and Integer operations are part of the set of functions $\mathcal{F}_K$. Expressions used in this thesis are first-order logic expression.

Definition 2.9 (First-order Formulas)**.** Let K be a knowledge base with types $\mathcal{T}_K$, predicates $\mathcal{P}_K$, and functions $\mathcal{F}_K$. We define the *set of first order formulas over* K, named Φ_K, inductively as follows:

- if a, b are terms with $type(a) = T_1$ and $type(b) = T_2$, and $p : T_1 \times T_2 \in \mathcal{P}_K$, then $p(a, b) \in \Phi_K$;
- if $t_1, t_2 \in \Phi_K$, then $\neg t_1 \in \Phi_K$ and $t_1 \vee t_2 \in \Phi_K$;
- if $t \in \Phi_K$, then $\forall x : t \in \Phi_K$ and $\exists x : t \in \Phi_K$.

While variables in expressions are generally bound by quantifiers, we use input and output variables of service descriptions as free variables in expressions ($\to$ Section 3.2.1, Services and Service Compositions). For a formula $\varphi \in \Phi_K$, we use $\mathit{free}(\varphi)$ to denote the set of free variables of φ.

We name a fixation of variable values a *state* (Apt et al., 2009, p. 34). Later in this thesis, we use states to describe the control flow of service compositions.

Definition 2.10 (States)**.** For a set of typed variables Var with types $\mathcal{T}_K$ of a knowledge base K, a *state* is a mapping of variables to elements of their respective domain:

$$\begin{aligned} \sigma : \mathit{Var} &\rightarrow \mathcal{U}_K \\ \text{with } \sigma(a) &\in \mathcal{U}_{\mathit{type}(a)} \\ \text{and } a \in \mathit{Var}, \mathit{type}(a) &\in \mathcal{T}_K, \mathcal{U}_{\mathit{type}(a)} \subseteq \mathcal{U}_K \end{aligned}$$

Sometimes we need to describe sets of states depending on the possible valuations of variables. To do this, we use expressions in first-order logic as *assertions*. Assertions evaluate to true or false, depending on the valuation of variables used in the assertion, that is, depending on a state. We write $\sigma \models p$ if an assertion holds (evaluates to true) in a state σ. Assertions are evaluated in the usual way (e.g., Apt et al., p. 41-42), based on both a state (for variable values) and a logical structure (for predicate interpretations).

We also use assertions to define the set of states in which an assertion holds.

Definition 2.11 (Sets of States)**.** Let $p \in \Phi_K$ be an assertion. We define the *set of states* satisfying p with respect to a logical structure $\mathcal{S}$ as $[\![p]\!]_{\mathcal{S}}$:

$$[\![p]\!]_{\mathcal{S}} = \{\sigma \mid \sigma \models_{\mathcal{S}} p\}$$

Deciding whether a state satisfies an assertion, or, more interestingly, whether there exists a logical structure such that there exists a state to satisfy an assertion, can be automated using existing satisfiability solvers.

The following chapters build on this concept of a knowledge base and the accompanying definitions.

Part II

Contributions

Chapter 3

Workflow Descriptions

Workflows are descriptions of repetitive tasks (Smith, 2007). Imperative programming languages, flow charts, and business process modeling languages are examples on a scale of workflow modeling techniques in computer science, spanning a range from defining lowest-level steps of assigning values to memory registers, to abstract process executions triggering complex human-based processes in companies. Conceptually, workflow descriptions are about structuring the order and relation of single actions. On this level, they differ from programming languages mostly in their area of application, and therefore in abstract, business oriented contexts, we use the term *workflow*, and in programming contexts, we use *programming language*.

There are numerous approaches in computer science literature dealing specifically with workflow and process modeling, both in general and concerning business processes. If we add imperative programming languages, we have several ways to model a process consisting of single steps. Section 3.1 gives an overview. However, in the context of formalized domain knowledge, and with the goal of formal verification in mind, we have certain requirements to a workflow specification language which go beyond the mere structuring of actions. Adding the context of services and On-The-Fly Computing, we can identify three constraints:

At first, a *service call* as a single workflow step, or action, is at the core of a workflow, and therefore the base of inductive workflow definitions. It has to be mapped to a identifiable service which has to contain a formalized description of its semantics, i.e., its behavior. A service call is not only embedded in the control flow of a workflow, but also in the data flow: The source of its inputs has to be defined as well as the use of its outputs. We call this the need for *semantic service descriptions.*

At second, the workflow description itself has to have a formalized semantics. In order to support a formal verification, it has to be built upon the semantic description of services, and take the control flow as well as the data flow into account. Also, the semantics must support a notion of correctness as well as a means to prove or disprove whether or not a given workflow is correct. We call this the need for *rigorous workflow semantics.*

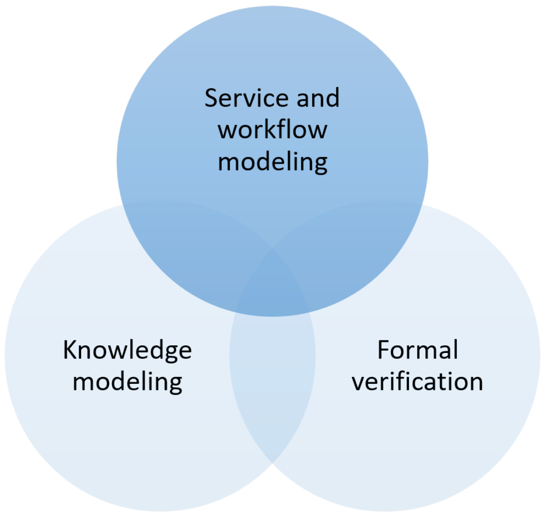

Figure 3.1: Joint research: Service and workflow modeling in relation to knowledge modeling and formal verification

At third, service composition takes place within a specialized business domain. As stated in Chapter 2, *formalized domain knowledge* provides syntax and semantics to formalize knowledge. Therefore, the formalization of service descriptions as well as the semantics of workflows have to rely on the same mechanisms as the knowledge semantics. This is the need to incorporate *formalized domain knowledge*.

As verification is built upon all three elements, a close relation between these three topics is mandatory to make verification results sound. Without these commonalities, neither domain knowledge, nor service descriptions, nor results of program verification can claim to be reliable in an on-the-fly context ($\rightarrow$ Figure 3.1).

Chapter 2 (Domain Knowledge and Logic) lays the logical foundation for the context of this thesis. In this chapter, we build on this foundation. First, we relate to the most common approaches of workflow- and service-based modeling in the area of computer science (Section 3.1), and identify a core notion of "service". To address the aforementioned requirements, we then define a workflow language based on service calls, consisting of the most common control flow structures. We add an operational semantics based on states in terms of variable valuations, and therefore continue the use of logical structures (Section 3.2). We define partial and total correctness for workflows specified in this language (Section 3.3) and provide a Hoare-style proof calculus as a tool to prove or disprove correctness of a workflow (Section 3.4). We also show that the calculus is sound and complete.

3.1 Related Work

Formal workflow modeling in a practical context joins the areas of programming, business process modeling, and service descriptions. On one end of the spectrum of workflow descriptions are programming languages. Assembler-level programming languages describe highly specialized workflows and are targeted at executing machine-specific calculations. High-level, general purpose languages are no longer machine-specific and aim at a more abstract programming approach. On top of that, domain specific languages try to focus more on the task at hand instead of the intricacies of general computer programming (van Deursen et al., 2000). These are, however, different levels of abstractions which share their origin in "making a computer calculate". They work in a highly restricted environment with a focus on their execution on a given machine. This does not necessarily mean that they have a formally defined semantics: More often than not, the semantics of a programming language is defined by an "execution semantics", that is, it depends on how the compiler, interpreter, and executing machine handle the language constructs. This leads to noteworthy behavior like different effects of the same language constructs on different execution environments. This results from a textual semantics specification, which may be backed by a reference implementation, e.g., for Java.[1]

Business process descriptions are located on the other end of the spectrum. They started with a focus on describing a workflow as a sequence of atomic actions and thus enabled the division of labor (Smith, 2007). Today, from an economic perspective, this makes complex processes not only repeatable, but also easier to understand, especially in the context of the organizational structure of a company (Davenport, 1998). Workflow management systems integrate a formal description, domain (or company) specific information, and organizational background of a company. Actions are atomic only with respect to a chosen abstraction level (van der Aalst and van Hee, 2002).

In computer science, process descriptions on a business level quite naturally incorporate software systems and data (or documents) which are moved by actors between these systems. While this can be expressed by flowcharts on both an informal (e.g., "boxes and arrows") and semi-formal (e.g., UML activity diagrams[2], Dumas and ter Hofstede 2001) level, the *web services business process execution language* (WSBPEL) as an OASIS standard is a major player.[3] WSBPEL formalizes executable workflows (or processes) which use Web services as its atomic actions. While the WSBPEL specification does not contain a formal semantics, several approaches exist to define a semantics for verification purposes, e.g., based on abstract state machines (Farahbod et al., 2005), or on translation to Petri nets (Ouyang et al., 2007). Several workflow execution engines exist to run BPEL workflows, e.g., Microsofts BizTalk[4], Oracle's BPEL Process Manager[5],

[1]Oracle's Java SE specifications: `https://docs.oracle.com/javase/specs/`, retrieved July 21, 2017.

[2]OMG's UML specification: `http://www.omg.org/spec/UML/`, retrieved July 21, 2017.

[3]OASIS' WSBPEL specification: `https://www.oasis-open.org/committees/wsbpel`, retrieved July 21, 2017.

[4]`https://www.microsoft.com/en-us/cloud-platform/biztalk`, retrieved July 21, 2017.

[5]`http://www.oracle.com/technetwork/middleware/bpel/overview/index.html`, retrieved July 21, 2017.

or Apache's Orchestration Director Engine[6]. To be usable in a BPEL description, Web services descriptions have to be given in the *web service description language* (WSDL, Weerawarana et al. 2007).

A formal service description in a programming sense exists also for Web-based software, or Web services. The Service-Oriented-Architecture approach (SOA) aims at a rich description of Web services (or services in general), to leverage their integration into an, possibly executable, workflow (van der Aalst et al., 2006). The Web Service Description Language (WSDL) emerged as an W3C[7] standard (Weerawarana et al., 2007). While it already enables a syntactical description of services and how to call them, a real advantage is the addition of semantic annotations (SA-WSDL) to include logical pre- and postconditions in the description (Farrell and Lausen, 2007). The orchestration languange OWL-S goes a step further and is based on an ontology, that is, a knowledge base, specialized on the description of services (Martin et al., 2005). OWL-S combines an abstract, task-oriented view with a technical, orchestration-oriented view, where actual deployment of services and service compositions is critical.

Adding such semantic annotations is not restricted to the Semantic Web. In program verification, logical verification conditions are anything but new (Floyd, 1967). They are also present in modern languages, either built-in as in Eiffel (Meyer, 1997), or by extensions like the Java Markup Language (JML, Leavens, Baker, and Ruby 1999), ESC/Java2 (Flanagan et al., 2002), or Spec# (Barnett et al., 2005). However, either the vocabulary of these semantic annotations is quite restricted (which is the case in the field of program verification, where it is tailored to predicates concerning variable values), or the rigorous verification techniques are not applicable because of a lacking appropriate formal semantics.

So while there is a lot of research combining two of the three topics of workflow semantics, semantic service description, and formal knowledge base modeling, combining all of them in a theoretically sound way is still an open issue.

3.2 Syntax and Semantics

Core part of service compositions is the service *orchestration* or *workflow definition*, which comprises of the control and data flow between the composition's input, output, and the called services. Basic building blocks are calls to existing services, and the control structures themselves. In the context of On-The-Fly Computing we distinguish atomic, "black box" services and to complex, "gray/white box" service compositions. While it is possible that atomic services are compositions as well, they always include a service description, and can therefore be treated as atomic services. Additionally, we identify common control flow structures to compose atomic services into a composition.

[6]`http://ode.apache.org`, retrieved July 21, 2017.
[7]Word Wide Web Consortium, `http://w3c.org`, retrieved July 21, 2017.

Section 3.2.1 defines the syntax of service descriptions, workflows, and their relation to the domain knowledge vocabulary. Section 3.2.2 defines an operational semantics of workflows, before we continue with a definition of correctness in Section 3.3.

3.2.1 Services and Service Compositions

Every structured workflow is defined inductively, starting with atomic statements, and continuing with composed control structures. Calls to single services and variable assignments are atomic statements. From the modeling point of view, the relevant part of a service is its *service description*. Thinking in Web services, a colloquial "service" may consist of several different methods: A web shop service may consist of methods *login*, *browse*, *add to cart*, etc., where every method is backed by an actual implementation, that is, eventually, a method call. The overall collection of these "granular" methods is considered to represent "the" web shop service.

In order to call a service, a description of its interface is necessary. From a programming point of view, this implies the use of method signatures, including naming (or addressing in general) as well as number and type of input and output parameters. Additionally, a description elaborates the behavior of the method and how to use it. For programming languages, this description typically comes as *application programming interface* (API), as for Java[8] and C#[9]. Languages with a more formalized approach, e.g., OWL-S with SAWSDL for the semantic web, or JML, ESC/Java, or Spec# for programming languages, provide a logic-based language to formalize the semantic description of its behavior.

To simplify things, we restrict ourselves to *state-less services* (in terms of non-observable, internal states), that is, regardless of how many methods a service offers, every single one just processes input data and returns output data. This is consistent with, e.g., the REST (*representational state transfer*) approach of Web services, where state changes are always explicit (Fielding and Taylor, 2000). This restriction allows us to simplify our notion of "services": As a service (in the sense of Web service) is a collection of methods (or fine-granular "services") *which are mutually independent*, we can (a) assume that a service has a unique name, and (b) treat every method of a (colloquial) service as a complete (formal) service for its own sake.

Uniqueness of names can be guaranteed using appropriate names (and namespaces, as it is done with URIs). Therefore, the most basic requirement to a service description, in addition to the unique name, is the number (and order) of input and output elements. As we operate in the context of a formalized description of domain knowledge, we can rely on formalized types ($\rightarrow$ Definition 2.5, Types). Based on types, we define a *service signature*, which essentially defines the signature of the *function* which the service represents in a mathematical sense.

[8]`https://docs.oracle.com/javase/8/docs/api/index.html`, retrieved July 21, 2017.
[9]`https://docs.microsoft.com/en-us/dotnet/api/`, retrieved July 21, 2017.

Definition 3.1 (Service Signature)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, with $\mathcal{T}_K$ the set of types associated with this domain. Then

$$Svc : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_n$$

is the *service signature* Svc^{Sig} of the service named Svc, with input types $T_1, \ldots, T_k$, and output types $T_{k+1}, \ldots, T_n$, where $T_i \in \mathcal{T}_K$ for $1 \leq i \leq n$.

We combine this mathematical signature with a logical description of input and output data. On the theoretical side, this dates back to Hoare, while on the practical side, it became more widespread with the advent of the *Semantic Web* (Berners-Lee et al., 2001) and service description languages like SAWSDL (Farrell and Lausen, 2007) and OWL-S (Martin et al., 2005). In program verification, it is used in the presence of function calls (JML, Spec#) and code variant verification (e.g., Soleimanifard and Gurov, 2015).

With a mathematical signature alone, it is not possible to use predicates for description of the input and output of a service. The predicates need *parameters* to refer to the exact data handled by the service. As in programming languages and service description languages we name the variable parameters and use the same names as parameters in pre- and postconditions, and therefore extend the mathematical service signature with a *service description.*

Definition 3.2 (Service Description)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, with $\mathcal{T}_K$ the set of types associated with this domain. Let $S^{Sig} = S : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_n$ be the signature of a service S. Then

$$S^{Desc} = (S^{Sig}, I_S, O_S, pre_S, post_S)$$

denotes the *service description* of S, with

- the service signature S^{Sig},
- input variables I_S with $|I_S| = k$,
- output variables O_S with $|O_S| = (n - k)$,
- a precondition $pre_S \in \Phi_K$ with $free(pre_S) \subseteq I_S$,
- a postcondition $post_S \in \Phi_K$ with $free(post_S) \subseteq (I_S \cup O_S)$,
- and $I_S \cap O_S = \emptyset$.

We assume I_S and O_S to be ordered such that variable names uniquely correspond to a position in S^{Sig}. We denote the set of all services of a domain with knowledge base K as $\mathcal{SVC}_K$.

In resemblance of function specifications in programming languages on the one hand, and of Hoare-triples on the other hand, we write a service description with conditions in curly brackets, and the signature with additional variable names accompanying the variable types:

$$\begin{aligned}
&\{pre(i_1, \ldots, i_k)\}\\
&Svc(i_1 : T_1, \ldots, i_k : T_k, o_{k+1} : T_{k+1}, \ldots, o_n : T_n)\\
&\{post(i_1, \ldots, i_k, o_{k+1}, \ldots, o_n)\}
\end{aligned}$$

Example 3.1 (Service: Rating Acquisition)**.** *Assume a service in the Tourism domain, which makes use of the domain knowledge described in Example 2.1. The service* Get-Rating *is a lookup service to provide a* Rating *(which is a concept of the domain, and therefore also a type generated by the domain) for a given* Restaurant*. It therefore has the following signature:*

$$GetRating : Restaurant \to Rating$$

Its description contains one input variable and one output variable, whose names can be used in the pre- and postconditions. The service does not have a precondition. As a postcondition, it guarantees that the resulting rating actually belongs to the given restaurant,using the predicate hasRating*. In tuple notation, we write*

$$\big((GetRating : Restaurant \to Rating), \{restaurant\}, \{rating\},\\ \top, hasRating(restaurant, rating)\big) \ .$$

In triple notation, we write

$$\begin{aligned}
&\{\}\\
&GetRating(restaurant : Restaurant, rating : Rating)\\
&\{hasRating(restaurant, rating)\} \ .
\end{aligned}$$

For a service S, we use its *name* to denote the name S itself, the signature S^{Sig} of which it is an integral part, and to refer to the overall service description S^{Desc}. Later, it will also be used as part of the control flow syntax. This way, we avoid using different variants of the name to refer to just one service.

In programming, assignment and function calls are basic building blocks of programs. Analogously, we inductively define a workflow language using the usual control flow elements, with service calls and variable assignment as induction base cases. Later on, we need to precisely identify the “position” in a workflow to show correctness automatically. To this end, we *label* every workflow statement in a workflow, using a unique label l. We will come back to these labels only in Chapter 5 (Automating Correctness Proofs using First-order Logic).

Definition 3.3 (Workflow)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. The following rules define the syntax of a *workflow* W:

$$\begin{aligned} W ::= \; & [l] \; \mathit{skip} \\ \mid \; & [l] \; u := t \\ \mid \; & W_1 ; W_2 \\ \mid \; & [l] \; (u_{j+1}, \ldots, u_k) := S(i_1, \ldots, i_j) \\ \mid \; & [l] \; \texttt{if } B \texttt{ then } W_1 \texttt{ else } W_2 \texttt{ fi} \\ \mid \; & [l] \; \texttt{while } B \texttt{ do } W_1 \texttt{ od} \\ \mid \; & [l] \; \texttt{foreach } a \in A \texttt{ do } W_1 \texttt{ od} \end{aligned}$$

with $u, v, a, A \in \mathit{Var}$, $t \in \Phi_K$, $\mathit{type}(t) = \mathit{type}(u)$, $\mathit{type}(A) = \textbf{set } T$, $\mathit{type}(a) = T, T \in \mathcal{T}_K$, $B \in \Phi_K$, and $S \in \mathcal{SVC}_K$. Additionally, we define exactly one final label $[\mathit{end}]$ for a workflow W.

While *Var* denotes the set of all variables, we need to refer to free, that is, input variables of a workflow as well as variables that are changed by a workflow.

Definition 3.4 (Variables of Workflows)**.** Let *Var* be the set of all variables and $\mathit{Var}(W) \subseteq \mathit{Var}$ the set of variables which appear in a workflow W. Then $\mathit{change}(W) \subseteq \mathit{Var}(W)$ denotes the set of variables which appear on the left-hand side of an assignment or service call statement. The set of free variables of W is $\mathit{free}(W) = \mathit{Var}(W) \setminus \mathit{change}(W)$.

We denote the empty workflow as E with the following properties:

$$\begin{aligned} E ; E &\equiv E \\ E ; W &\equiv W \\ W ; E &\equiv W \end{aligned}$$

A workflow description is the core of a service composition. In an on-the-fly computing context service compositions are created up to a predefined signature and pre- and postcondition. The mere *description* of a service composition is therefore exactly the same as the description of an atomic service. We define a complete service composition as follows.

Definition 3.5 (Service Composition)**.** A *service composition* (Sc^{Desc}, W) consists of a service description $Sc^{Desc} = (Sc^{Sig}, I_{Sc}, O_{Sc}, \mathit{pre}_{Sc}, \mathit{post}_{Sc})$, and a workflow W, with $\mathit{free}(W) \subseteq I_{Sc}$, $O_{Sc} \subseteq \mathit{change}(W)$, and $I \cap O = \emptyset$.

The variable restrictions enforce a workflow structure similar to *static single assigment* (SSA) form and therefore guarantee that input variables are never written to, and, consequently, output variables can always be compared to original inputs without additional computations of data flow graphs (Cytron et al., 1991). While this may seem restrictive at first, this semi-SSA form can be achieved easily by introducing fresh variables for input variables, as we require "write once" only for inputs (and not for all free variables) and therefore avoid the necessity of phi functions to model alternative assignments resulting from loops.

3.2.2 States, Configurations, and Semantics of Workflows

Based on a syntax definition of service descriptions and service compositions, we want to define a "correct" composition, and we want to be able to formally *reason* about it. To this end, we first need to define what we actually mean when a workflow is executed. That is, we need a formally defined semantics. The need for a formalized semantics, in contrast to an intuitive understanding of the meaning of a program, was proposed by Floyd as early as 1967 (Floyd, 1967). He introduced an *interpretation* of syntax elements of programming languages to propositions which hold before and after the basic syntactic elements. He demonstrates his semantics with a graphical flowchart language and a subset of ALGOL.

From there, two main approaches to formalize semantics developed. The *denotational* approach defines the semantics of a program by associating program statements with mathematical functions (Scott, 1972). The semantics are then defined by choosing domains (and ranges) for these functions, and restricting their interpretation by mathematical equations. The *operational* approach, on the other hand, focuses on explicit state changes caused by program statements (Apt et al., 2009, Plotkin, 1981, 2004). Here, a program state is represented by a given valuation of variables, and each program statement modifies a given state as defined by a corresponding transition axiom (or rule). This view on actual state changes has a close connection to actually *interpreting* a program. Additionally, an *axiomatic* approach defines semantics based on Hoare-style rules (Hoare, 1969). As Plotkin points out, there is a useful connection between an operational and an axiomatic semantics definition: While axiomatic notations are elegant and precise, operational semantics with their view on program states can be used to show that axiomatic systems are sound and complete.

This is how Apt, de Boer, and Olderog simplify reasoning about the correctness of programs, and why we follow their approach in this thesis. To this end, we proceed as follows: Based on Apt et al.'s operational semantics for the WHILE language, we define an *operational semantics* for workflows to relate the "before" state of a workflow statement with its "after" state. The state of a program is represented by the valuation of its variables at a given workflow location. This location can be tracked with a dedicated program counter variable. In Definition 3.3 (Workflow), we identify locations using labels, but labels will not become relevant before an actual state encoding in Chapter 5. However, a valuation of variables always depends on a logical structure: The structure determines the universes of the variable types, and the exact interpretation of not only default operations, but also domain-specific predicates. To decide whether a formula is satisfied in a given state σ, e.g., $\sigma \models (x > y) \wedge Q(x, y)$, the interpretation of predicates in addition to a mere valuation of variables is crucial.

As predicates with no exactly defined interpretation are the main ingredient of ontologies, and therefore knowledge bases, our definitions cannot rely on "standard" interpretations (as for Integer operations) alone. Therefore, a valuation of a formula in a given state has to consider the actual logical structure. In contrast to Apt et al., we do not fix the logical structure (Apt et al., 2009, p. 33). Instead, it has to be a *parameter* of the valuation. Without fixing $\mathcal{S}$, we can only ask the question whether a state σ satisfies a formula *for a given structure* $\mathcal{S}$:

$$\sigma \models_{\mathcal{S}} (x > y \wedge Q(x, y))$$

This *parameterization* enables us to reason about states which satisfy a formula *for all possible structures*. The definition of correctness of service compositions in Section 3.3, p. 42 relies on this fact. Parameterizing logical satisfiability with a logical structure and then quantifying over all structures seems to be a complicated way to write "tautology". However, later on we will quantify only over selected structures, and therefore this parameterization enables an elegant formulation of later definitions.

We combine states and workflows to define an operational semantics for workflows. As a workflow is composed of different statements, we can always relate a position in a workflow with a state σ. Following Apt et al., we define a *configuration* to represent a state and the workflow that remains to be executed immediately after the given state (Apt et al., 2009, p. 58).

Definition 3.6 (Configuration)**.** Let σ be a state, and let W be a workflow. Then,

$$\langle W, \sigma \rangle$$

denotes a *configuration*, with state σ and workflow W that remains to be executed.

We denote a *transition* between configurations (and therefore states) as $\langle W_1, \sigma_1 \rangle \rightarrow_{\mathcal{S}} \langle W_2, \sigma_2 \rangle$: When we execute W_1, starting in state σ_1, then we reach state σ_2 and workflow W_2 is the remaining workflow to be executed. As a transition connects states, which are represented by valuations of variables, it has to be parameterized with the logical structure. Depending on potential transition sequences, a workflow may *terminate*, *block*, or *diverge*. The definition follows Apt et al., Def. 3.1, p. 59.

Definition 3.7 (Termination, Blocking, and Divergence of Workflows)**.** A *transition sequence* is a finite or infinite sequence of configurations $\langle W_1, \sigma_1 \rangle \rightarrow_{\mathcal{S}} \cdots \rightarrow_{\mathcal{S}} \langle W_i, \sigma_i \rangle \ldots$ A *computation* of a workflow W starting in σ is a transition sequence which either (a) cannot be extended, or (b) is infinite. A computation *terminates*, if it cannot be extended and ends with $\langle E, \tau \rangle$, that is, the empty workflow. A computation *blocks*, if it cannot be extended, but ends with $\langle W', \tau \rangle$ and $W' \not\equiv E$. A computation *diverges*, if it is infinite.

We define our semantics by defining state transitions inductively for every workflow statement. For readability, we assume that services only have exactly one input and one output, that is, we use

$$u := S(v)$$

instead of

$$(u_1, \ldots, u_i) := S(v_1, \ldots, v_k)$$

for calling a service S. We assume this only for a simplified notation with generic services, not in general.

Definition 3.8 (Transition Axioms and Rules for Workflows)**.** For a knowledge base K, let $u, v, a, A \in \mathit{Var}$, $t \in \Phi_K$, $\mathit{type}(t) = \mathit{type}(u)$, $\mathit{type}(A) = \textbf{set } T, \mathit{type}(a) = T, T \in \mathcal{T}_K$, $B \in \Phi_K$, and $\mathit{Svc} \in \mathcal{SVC}_K$. We define the following transition axioms and rules for a workflow W and a state σ as follows:

Skip	$\langle \mathit{skip}, \sigma\rangle$	$\rightarrow_S \langle E, \sigma\rangle$
Assignment	$\langle u := t, \sigma\rangle$	$\rightarrow_S \langle E, \sigma[u := \sigma(t)]\rangle$
Take	$\langle \mathit{take}(a, A), \sigma\rangle$	$\rightarrow_S \langle E, \sigma'\rangle$
if		$\sigma'(a) \in \sigma(A), \sigma'(A) = \sigma(A) \setminus \{\sigma'(a)\}$,
and		$\sigma(x) = \sigma'(x)$ for $x \neq A, x \neq a$
Sequential comp.	$\dfrac{\langle W_1, \sigma\rangle \rightarrow_S \langle W_2, \tau\rangle}{\langle W_1; W, \sigma\rangle \rightarrow_S \langle W_2; W, \tau\rangle}$	
Cond. (then)	$\langle \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}, \sigma\rangle \rightarrow_S \langle W_1, \sigma\rangle$	
if		$\sigma \models_S B$
Cond. (else)	$\langle \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}, \sigma\rangle \rightarrow_S \langle W_2, \sigma\rangle$	
if		$\sigma \models_S \neg B$
While (loop)	$\langle \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma\rangle$	$\rightarrow_S \langle W; \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma\rangle$
if		$\sigma \models_S B$
While (end)	$\langle \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma\rangle$	$\rightarrow_S \langle E, \sigma\rangle$
if		$\sigma \models_S \neg B$
Foreach (loop)	$\langle \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma\rangle$	
		$\rightarrow_S \langle \mathit{take}(a, A); W; \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma\rangle$
if		$\sigma \models_S A \neq \emptyset$
Foreach (end)	$\langle \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma\rangle$	$\rightarrow_S \langle E, \sigma\rangle$
if		$\sigma \models_S A = \emptyset$
Service call	$\langle u := \mathit{Svc}(v), \sigma\rangle \rightarrow_S \langle E, \sigma'\rangle$	
if		$\sigma \models_S \mathit{pre}_{\mathit{Svc}}[i := v]$,
		$\sigma' \models_S \mathit{post}_{\mathit{Svc}}[i := v, o := u]$,
and		$\forall x \neq u : \sigma'(x) = \sigma(x)$

In this definition, the exact transitions rely on the underlying logical structure, especially on the interpretation of predicates from the knowledge base. The `if` clause is an example: The next workflow is determined by the evaluation of the clause's condition (B). This condition may contain an expression with predicates from the knowledge base, and therefore its interpretation depends on the logical structure. As we do not fix a logical structure, it has to be a parameter in the definition of the semantics.

This definition gives us foundations for a workflow semantics. We use the *transitive closure* $\rightarrow^*_{\mathcal{S}}$ to denote concatenations of state transitions.

3.3 Partial and Total Correctness

Now, the semantics has to map a state *before* executing a workflow to a state (or a set of possible states) *after* the workflow. Depending on how we treat non-termination, we define a partial and a total correctness semantics.

The semantics of a workflow is a mapping from a state to the corresponding states which are reached by applying Definition 3.8 (Transition Axioms and Rules for Workflows) until the workflow terminates, that is, only the empty workflow E remains. Workflows do not necessarily terminate. Non-terminating workflows either *diverge* (e.g., non-terminating loops as in `while true do` W `od`), or *block*. Workflows can block if the precondition of a service evaluates to false. In that case, no transition can take place, as Definition 3.8 does not contain a corresponding transition rule. Therefore, correctness semantics may consider all possible workflows (total correctness) or only (successfully) terminating workflows (partial correctness). We define correctness semantics for both cases, and denote the semantics of a workflow W using the semantic parentheses. We follow the Definitions 3.2 (i) and (ii) from Apt et al. (2009).

Definition 3.9 (Partial Correctness Semantics of Workflows)**.** Let W be a workflow and $\sigma \in \Sigma$ be a state. We define the *partial correctness semantics* as a mapping

$$[\![W]\!]_{part,\mathcal{S}} : \Sigma \rightarrow 2^{\Sigma}$$

with

$$[\![W]\!]_{part,\mathcal{S}}(\sigma) = \{\tau \mid \langle W, \sigma\rangle \rightarrow^*_{\mathcal{S}} \langle E, \tau\rangle\} \ .$$

A complementary definition of *total correctness* includes workflows which do not terminate. To do so, we introduce a *failure state* $\bot$ to denote both diverging and blocking executions of a workflow.[10] In a definition of total correctness, this failure state is a possible end-state of a sequence of transitions.

Definition 3.10 (Total Correctness Semantics of Workflows)**.** Let W be a workflow and $\sigma \in \Sigma$ be a state. We define the *total correctness semantics* as a mapping

$$[\![W]\!]_{tot,\mathcal{S}} : \Sigma \rightarrow 2^{\Sigma \cup \{\bot\}}$$

with

$$[\![W]\!]_{tot,\mathcal{S}}(\sigma) = [\![W]\!]_{part,\mathcal{S}}(\sigma) \cup \{\bot \mid W \text{ diverges or blocks on } \sigma \text{ w.r.t. } \mathcal{S}\} \ .$$

[10]In logical formulas, we still use $\bot$ to denote `false`, as we use $\top$ to denote `true`.

For easier reading, we use $[\![W]\!]_{\mathcal{S}}$ to refer to both the partial and total correctness semantics, unless a distinction is necessary. We lift the definition of correctness semantics from a mapping of single states to a mapping of sets of states.

Definition 3.11 (Correctness Semantics for Sets of States)**.** Let W be a workflow and $\sigma \in \Sigma$ be a state, and let p and q be assertions. We define the *partial and total correctness semantics on sets of states* as

$$[\![W]\!]_{\mathcal{S}} ([\![p]\!]_{\mathcal{S}}) = \{\tau \mid \sigma \in [\![p]\!]_{\mathcal{S}} \wedge \tau \in [\![W]\!]_{\mathcal{S}} (\sigma)\} \ .$$

With a formally defined semantics, we proceed to define a “correct” composition. When we consider a service composition (or any other service, for that matter) as *correct*, we always do so in the sense that it is *correct with respect to its pre- and postconditions.* Correctness in this sense is the guarantee of providing the postcondition after executing the composition, as long as the precondition holds at the beginning of the execution. The formal definition of correctness reflects that. As we do not fix a logical structure, the definition includes quantification over structures.

Definition 3.12 (Correctness of Compositions)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. Let (Sc^{Desc}, W) be a service composition with a description $Sc^{Desc} = (Sc^{Sig}, I_{Sc}, O_{Sc}, pre_{Sc}, post_{Sc})$. The service composition is *correct with respect to its pre- and postcondition*, if and only if

$$\forall \mathcal{S} \text{ with } \mathcal{S} \models K : \quad [\![W]\!]_{\mathcal{S}} ([\![pre_{Sc}]\!]_{\mathcal{S}}) \subseteq [\![post_{Sc}]\!]_{\mathcal{S}} \ .$$

Correctness is always partial or total correctness (depending on the semantics). Following Hoare-style notation, we write $\models_K \{pre_{Sc}\}\, W\, \{post_{Sc}\}$ to denote that a workflow W is correct with respect to this definition under K.

For easier reading, we will sometimes use the name of the service composition instead of the workflow, that is, instead of

$$\begin{aligned}&\models_K \{pre_{Sc}\}\, W\, \{post_{Sc}\} \text{ for } (Sc^{Desc}, W)\\ &\text{with } Sc^{Desc} = (Sc^{Sig}, I_{Sc}, O_{Sc}, pre_{Sc}, post_{Sc}),\end{aligned}$$

we just write $\models_K \{pre_{Sc}\}\, Sc\, \{post_{Sc}\}$.

3.4 Proof Calculus

We defined correctness of workflows based on its semantics. Now, to prove correctness of a given workflow, we have to find chains of transitions from the starting state(s) to the end state(s). While this approach corresponds with the idea of “interpreting” a workflow, the resulting proofs are based on an unfamiliar, configuration based notation, and one needs knowledge of the semantic formalisms to read and understand them.

In contrast, *syntax*-based proofs do not require this additional understanding. Hoare designs his famous syntax-based proof calculus based on two principles: He denotes which assumptions are true *before* and *after* a given program (or workflow) statement, and how these assumptions change for the various statements of a language. To this end, he defines proof axioms and rules (Hoare, 1969). Owicki and Gries integrate the resulting proofs directly into the program, alternating statements and *assertions*, resulting in readable *proof outlines* (Owicki and Gries, 1976). As the calculus is syntax-based, it is necessary to prove a connection between a proof (outline) and the actual semantics-based definition of correctness by showing that the calculus is sound and complete.

In this section, we adapt the Hoare calculus to our workflow language. Adapting the calculus is common in various areas of research. Examples include the verification of abstract state machines with control states (Gabrisch and Zimmermann, 2012), synchronous languages for reactive systems (Gesell and Schneider, 2012), or concurrent programs with higher-order concurrency (Turon et al., 2013). With our approach, we follow Apt, de Boer, and Olderog (2009).

3.4.1 Axioms and Rules

At the core of the calculus are *proof axioms* and *proof rules.* There is one axiom or one rule for every different workflow statement, to enable us to reason about every possible workflow. We use the rules given in Apt et al. (2009) as a base and adapt them as necessary to our workflow language. Modifications include not only additional language constructs (namely the service call and set iteration), but especially the parameterization with logical structures to introduce the dependency to some given domain knowledge.

Definition 3.13 (Parameterized Proof Calculus for Workflows)**.** For a knowledge base K, let $p, q, t, \mathit{pre}, \ \mathit{post} \in \Phi_K, \ i, o, u, v, w, a, A \in \mathit{Var}, \ \mathit{type}(t) = \mathit{type}(u), \ \mathit{type}(A) = \textbf{set } T, \mathit{type}(a) = T, T \in \mathcal{T}_K, \ B \in \Phi_K, \ \mathit{Svc} \in \mathcal{SVC}_K$, and W, W_1 and W_2 workflows. Then the following axioms and rules define a *parameterized proof calculus for workflows.*

(1) Skip

$$\frac{}{\{p\}\, \mathit{skip}\, \{p\}}$$

(2) Assignment

$$\frac{}{\{p[u := t]\}\, u := t\, \{p\}}$$

(3.1) Service Call (for partial correctness)

$$\frac{}{\{\forall w \text{ s.t. } \mathit{post}_{\mathit{Svc}}[i := v, o := w] : q[u := w]\}\, u := \mathit{Svc}(v)\, \{q\}}$$

(3.2) Service Call (for total correctness)

$$\frac{}{\{\mathit{pre}_{\mathit{Svc}}[i := v] \wedge \forall w \text{ s.t. } \mathit{post}_{\mathit{Svc}}[i := v, o := w] : q[u := w]\}\, u := \mathit{Svc}(v)\, \{q\}}$$

(4.1) Take (for partial correctness)

$$\frac{}{\{\forall b \in A : q[a := b, A := A \setminus \{b\}]\}\, \mathit{take}(a, A)\, \{q\}}$$

(4.2) Take (for total correctness)

$$\frac{}{\{A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]\}\, \mathit{take}(a, A)\, \{q\}}$$

(5) Sequential composition

$$\frac{\{p\}\, W_1\, \{r\}, \{r\}\, W_2\, \{q\}}{\{p\}\, W_1; W_2\, \{q\}}$$

(6) Conditional

$$\frac{\{p \wedge B\}\, W_1\, \{q\}, \{p \wedge \neg B\}\, W_2\, \{q\}}{\{p\}\, \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}\, \{q\}}$$

(7.1) While (for partial correctness)

$$\frac{\{p \wedge B\}\, W\, \{p\}}{\{p\}\, \textbf{while}\ B\ \textbf{do}\ W\ \textbf{od}\, \{p \wedge \neg B\}}$$

(7.2) While (for total correctness)

$$\frac{\begin{array}{l}\{p \wedge B\}\, W\, \{p\},\\ \{p \wedge B \wedge t = z\}\, W\, \{t < z\},\\ p \Rightarrow t \geq 0\end{array}}{\{p\}\, \textbf{while}\ B\ \textbf{do}\ W\ \textbf{od}\, \{p \wedge \neg B\}}$$

(8) Foreach

$$\frac{\{p \wedge A \neq \emptyset\}\, take(a, A); W\, \{p\}}{\{p\}\, \textbf{foreach}\ a \in A\ \textbf{do}\ W\ \textbf{od}\, \{p \wedge A = \emptyset\}}$$

(9) Consequence

$$\frac{\models_K (p \Rightarrow r), \{r\}W\{s\}, \models_K (s \Rightarrow q)}{\{p\}W\{q\}}$$

Depending on whether partial or total correctness of a workflow are relevant, there are two alternative proof rules for service calls, the *take* statement, and `while` loops. The variants (3.1), (4.1), and (7.1) are sufficient to proof partial correctness, as they consider only terminating workflows. Rule (7.1) is also applicable to non-terminating loops, which corresponds with the definition of partial correctness. To prove total correctness, termination of a loop is relevant, and therefore part of the premise of rule (7.2). Here, t is an integer expression and z an integer variable (Apt et al., 2009, p. 71). For total correctness it is also relevant whether or not a service call blocks (rule 3.2) and a set of elements of a loop actually contains anything (rule 4.2). In contrast, rule (8) is sufficient to prove both partial and total correctness for `foreach` loops. From the semantics of `foreach` loops and the *take* statement, we can see that termination is always guaranteed, as the number of loop executions equals the number of elements of the (finite) set. This corresponds to the expected behavior of iterator loops in imperative programming.

There are two axioms and one rule which can be considered special. The *take* axioms do not refer to a language construct. The *take* element is only part of the semantics of the iterator loop, and it serves as a helper construct to do loop unrolling and to iterate over a finite set of elements. As it is part of the semantics definition, it must also be part of formal reasoning, and therefore it needs proof axioms. The *rule of consequence* allows for the concatenation of single proofs by strengthening and weakening formulas that hold before or after a workflow statement. It is the only rule that actually needs to evaluate the formulas in question and does not work on a mere syntactical level. Therefore, its definition is parameterized with a knowledge base to include every possible logical structure whose interpretation conforms with the knowledge base itself.

As we are not interested in general validity (tautology) of formulas, but only in validity in the context of a given domain and its formalized domain knowledge, we formalize its restriction by a knowledge base K.

Definition 3.14 (Restricted Validity)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base, and let $p \in \Phi_K$. We call *p valid under K*, if it is valid (a tautology) for every logical structure $\mathcal{S}$ which adheres to the knowledge base, that is, the following holds:

$$\forall \mathcal{S} : \mathcal{S} \models R \Rightarrow \mathcal{S} \models p \ .$$

In short, we write $\models_K p$.

It is this kind of restricted validity which we employ in the rule of consequence. It guarantees validity in every possible logical structure (esp. its interpretation of predicates), but restricted to interpretations which are valid in the context of the rules of K. Because of the rule of consequence, a proof created with the proof calculus always works just for a given knowledge base K.

Definition 3.15 (Provable Properties)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base, and (Sc^{Desc}, W) a service composition. Then

$$\vdash_K \{p\}\, W\, \{q\}$$

denotes that there exists a proof outline for W under K.

Similar to the correctness notation, we use the name of the composition and write $\vdash_K \{p\}\, Sc\, \{q\}$ for short. We will use this definition to relate provability with correctness in a semantic sense.

3.4.2 Soundness

We defined an operational semantics based on the valuation of variables for workflows of service compositions. However, the proof calculus is defined on a syntactical level. The rational is to simplify proofs of correctness: If a proof is possible on a syntactical level using proof outlines, there is no need to argue using the semantics directly.

An important relationship between a syntax based proof and the semantically defined *correctness* of a workflow is the *soundness property* of the calculus: Whenever a workflow can be *proved* to be correct using the proof calculus, it has to be guaranteed that it actually *is* correct with respect to the definition of correctness (as in Definition 3.12). Correctness is based on the semantics, while the proof calculus as a proof tool is based on syntax. Therefore, we state a theorem to establish this relationship. We first published a variation of this theorem as well as its proof in Walther and Wehrheim (2015).[11]

[11]More precisely: We published the template version of this theorem, cf. Definition 4.15.

Theorem 3.16 (Soundness of Proof Calculus)**.** *Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base and (Sc^{Desc}, W) with $Sc^{Desc} = (Sc^{Sig}, I_{Sc}, O_{Sc}, pre_{Sc}, post_{Sc})$ a service composition. If Sc can be proved to be correct using the proof calculus, then it is correct according to the definition of correctness (→ Definition 3.12, Correctness of Compositions):*

$$\vdash_K \{pre_{Sc}\}\, W \,\{post_{Sc}\} \quad \Rightarrow \quad \models_K \{pre_{Sc}\}\, W \,\{post_{Sc}\}$$

We prove soundness of the calculus by induction, with axioms as base cases and rules as induction steps, that is, we show that axioms are true in the semantics, and the correctness of the premise of rules imply the correctness of their conclusion. As we follow Apt et al. (2009) with this structure, and as we extend their calculus with axioms and rules for service calls, *take*, and `foreach` statements, and the modified rule of consequence, we will only treat these here. To make the following proofs more readable, we use service calls with exactly one input and one output.

Proof of Soundness of Proof Calculus. Let K be a knowledge base, $Svc \in \mathcal{SVC}_K$, $A, a, b, u, v, w \in Var$, A of a set type, and $q, t \in \Phi_K$.

Service Call

For total correctness, we need to show that

$$\llbracket\, u := Svc(v) \,\rrbracket_{\mathcal{S}} \;(\, \llbracket\, pre_{Svc}[i := v] \wedge \forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w] \,\rrbracket_{\mathcal{S}} \,) \\ \subseteq \llbracket\, q \,\rrbracket_{\mathcal{S}}$$

holds. As a reminder, the conditions of the transition rule are:

$$\sigma \models_K pre_{Svc}[i := v] \tag{3.2}$$

$$\sigma' \models_K post_{Svc}[i := v, o := u] \tag{3.3}$$

$$\forall x \in Var \setminus \{u\} : \sigma(x) = \sigma'(x) \tag{3.4}$$

For easier reading, let

$$p := \quad pre_{Svc}[i := v] \wedge \forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w] \;.$$

Now, let σ, σ' be states with $\sigma \in \llbracket\, p \,\rrbracket_{\mathcal{S}}$ and $\langle u := Svc(v), \sigma\rangle \rightarrow_{\mathcal{S}} \langle E, \sigma'\rangle$.

By (3.3), we know that

$$\sigma' \models_K post_{Svc}[i := v, o := u] \tag{3.5}$$

and by (3.4) that the states are equal except for the valuation of u:

$$\exists w : \sigma' = \sigma[u := w] \;. \tag{3.6}$$

The valuation of u is not arbitrary because of (3.3), therefore we choose w such that

$$\sigma'(w) = \sigma'(u) \tag{3.7}$$

and therefore

$$\begin{aligned} & \sigma' \models_K post_{Svc}[i := v, o := u][u := w] \\ \Leftrightarrow \quad & \sigma' \models_K post_{Svc}[i := v, o := w] \ . \end{aligned}$$

Now, (3.6) and (3.5) lead to

$$\sigma[u := w] \models_K post_{Svc}[i := v, o := u]$$

and

$$\sigma \models_K post_{Svc}[i := v, o := u][u := w] \ .$$

By $\sigma \in [\![\, p \,]\!]$ and our knowledge about w in (3.7),

$$\sigma \models_K post_{Svc}[i := v, o := u][u := w] \wedge q[u := w]$$

and again

$$\sigma[u := w] \models_K post_{Svc}[i := v, o := u] \wedge q \ ,$$

therefore by (3.6),

$$\sigma' \models_K post_{Svc}[i := v, o := u] \wedge q$$

Then also

$$\sigma' \models_K q$$

and

$$\sigma' \in [\![\, q \,]\!]_{\mathcal{S}} \ .$$

For partial correctness, we need to show that

$$\begin{aligned} & [\![\, u := Svc(v) \,]\!]_{\mathcal{S}} \ ([\![\, \forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w] \,]\!]_{\mathcal{S}} \ ) \\ & \qquad \subseteq [\![\, q \,]\!]_{\mathcal{S}} \end{aligned}$$

holds. Otherwise, the proof is the same.

For total correctness, if $\sigma \in [\![\, p \,]\!]_{\mathcal{S}}$, then by (3.2), (3.3), and (3.4) there always exists σ' such that $\langle u := Svc(v), \sigma \rangle \to_{\mathcal{S}} \langle E, \sigma' \rangle$.

For partial correctness, if $\sigma \in [\![\forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w]]\!]_{\mathcal{S}}$ and $\sigma \notin [\![pre_S]\!]$, then $\langle u := Svc(v), \sigma\rangle \not\to_{\mathcal{S}} \langle E, \sigma'\rangle$, and the workflow blocks: $[\![u := Svc(v)]\!]_{part,\mathcal{S}}(\sigma) = \emptyset$, and $\emptyset \subseteq [\![q]\!]_{\mathcal{S}}$.

Take

For total correctness, we need to show that

$$[\![take(a, A)]\!]_{\mathcal{S}} ([\![p]\!]_{\mathcal{S}}) \subseteq [\![q]\!]_{\mathcal{S}}$$

with $p := A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]$. Let $\sigma \in [\![p]\!]_{\mathcal{S}}$ and $\langle take(a, A), \sigma\rangle \to \langle E, \sigma'\rangle$. By semantics of *take*, we get for any $b \in A$:

$$\sigma' = \sigma[a := b, A := A \setminus \{b\}] \ .$$

From $\sigma \in [\![p]\!]$, we get for all $b \in A$:

$$\sigma \models_K q[a := b, A := A \setminus \{b\}],$$

therefore, by the Substitution Lemma (Apt et al., 2009, Lemma 2.4, p. 47),

$$\sigma[a := b, A := A \setminus \{b\}] \models_K q$$

and

$$\sigma' \models_K q \ .$$

For partial correctness, we need to show the same with $p := \forall b \in A : q[a := b, A := A \setminus \{b\}]$, and the proof is the same.

For total correctness, if $\sigma \in [\![p]\!]_{\mathcal{S}}$, then by the condition of the transition rule for *take* there always exists σ' such that $\langle take(a, A), \sigma\rangle \to_{\mathcal{S}} \langle E, \sigma'\rangle$.

For partial correctness, if $\sigma \in [\![\forall b \in A : q[a := b, A := A \setminus \{b\}]]\!]_{\mathcal{S}}$ and $\sigma \notin [\![A \neq \emptyset]\!]_{\mathcal{S}}$, then $\langle take(a, A), \sigma\rangle \not\to_{\mathcal{S}} \langle E, \sigma'\rangle$, and the workflow blocks: $[\![take(a, A)]\!]_{part,\mathcal{S}}(\sigma) = \emptyset$, and $\emptyset \subseteq [\![q]\!]_{\mathcal{S}}$.

Foreach loop

We need to show that the premise of the `foreach` rule implies its consequence, that is:

$$\begin{aligned} &\models_K \{p \wedge A \neq \emptyset\}\; take(a, A); W\; \{p\} \\ \text{implies } &\models_K \{p\}\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}\, \{p \wedge A = \emptyset\} \end{aligned}$$

In contrast to `while` loops, `foreach` loops always have n executions of the loop body, with $|A| = n$, that is, the size of the set A, and thus always terminate. We include $|A| = n$ in the assertions. By the premise, the semantics of *take* and

$A \notin \mathit{change}(W)$, we get[12]

$$[\![\, \mathit{take}(a, A); W \,]\!]_{\mathcal{S}} \, ([\![\, p \wedge A \neq \emptyset \wedge |A| = n \,]\!]_{\mathcal{S}}) \subseteq [\![\, p \wedge |A| = n - 1 \,]\!]_{\mathcal{S}} \; . \tag{3.8}$$

We use this property to show that $[\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, p \,]\!]_{\mathcal{S}}) \subseteq [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}}$. The proof proceeds by induction.

Base case $|A| = n = 0$: By semantics of `foreach`, we have

$$[\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}}) = [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}} \; . \tag{3.9}$$

Induction step $|A| = n + 1$, and thus $A \neq \emptyset$. We use monotonicity of semantics for this proof (Apt et al., 2009, Lemma 3.3, p. 62-63).

$$\begin{aligned}
& [\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}}) \\
= & \; \{ \text{ by semantics of } \texttt{foreach} \text{ and } A \neq \emptyset \ \} \\
& [\![\, \mathit{take}(a, A); W; \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}}) \\
= & \; \{ \text{ by Def. of sequential composition } \} \\
& [\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, \mathit{take}(a, A); W \,]\!]_{\mathcal{S}}([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}})) \\
\subseteq & \; \{ \text{ by (3.8) and monotonicity of semantics } \} \\
& [\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, ([\![\, p \wedge \#A = n \,]\!]_{\mathcal{S}}) \\
\subseteq & \; \{ \text{ by induction hypothesis } \} \\
& [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}}
\end{aligned}$$

Rule of consequence By the premise of the rule, we have

$$[\![\, W \,]\!]_{\mathcal{S}} \, ([\![\, r \,]\!]_{\mathcal{S}}) \subseteq [\![\, s \,]\!]_{\mathcal{S}} \; ,$$

as well as valid implications $p \Rightarrow r$ and $s \Rightarrow q$ in K, therefore

$$[\![\, W \,]\!]_{\mathcal{S}} \, ([\![\, p \,]\!]_{\mathcal{S}}) \subseteq [\![\, W \,]\!]_{\mathcal{S}} \, ([\![\, r \,]\!]_{\mathcal{S}}) \subseteq [\![\, s \,]\!]_{\mathcal{S}} \subseteq [\![\, q \,]\!]_{\mathcal{S}} \; .$$

□

Thus, a proof outline constructed according to the proof calculus indeed shows correctness of its workflow.

[12]Actually, $A \notin \mathit{change}(W)$ has to be defined more precisely as in Definition 3.4 (Variables of Workflows). Otherwise, nesting loops ranging over the same set A does not work as expected. For convenience, we assume that a set can only be part of at most one `foreach` loop. In practice, that can easily be guaranteed by creating a fresh copy of A for every loop, instead of using A directly.

3.4.3 Completeness

A sound proof calculus guarantees that a (syntax based) proof of correctness obtained with the calculus is (semantically) correct. *Completeness* is its complementary property and shows that, if a workflow is correct, a proof actually exists.

Theorem 3.17 (Completeness of Proof Calculus). *Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, and let (Sc^{Desc}, W) with $Sc^{Desc} = (Sc^{Sig}, I_{Sc}, O_{Sc}, pre_{Sc}, post_{Sc1})$ be a service composition. If Sc is correct according to the definition of correctness (→ Definition 3.12), then correctness can be proved using the proof calculus:*

$$\models_K \{pre_{Sc}\}\, W\, \{post_{Sc}\} \quad \Rightarrow \quad \vdash_K \{pre_{Sc}\}\, W\, \{post_{Sc}\}$$

Proving completeness takes an intermediate step by defining *weakest liberal preconditions* and *weakest preconditions*. They are defined with respect to a set of target states which can be reached by executing a workflow W. The difference is their connection to partial and total correctness semantics: Weakest *liberal* preconditions are defined based on partial correctness semantics, while weakest (non-liberal, that is, strict) preconditions are based on total correctness semantics. We use the usual definition as in Apt et al., 2009, Def. 3.10, p. 86.

Definition 3.18 (Weakest (Liberal) Preconditions). Set W be a workflow and Φ be a set of states. We define the *weakest liberal precondition* of W regarding Φ as follows:

$$wlp(W, \Phi) = \{\sigma \mid [\![\, W \,]\!]_{part,\mathcal{S}}\,(\sigma) \subseteq \Phi\}\ .$$

Correspondingly, the *weakest precondition* of W regarding Φ is:

$$wp(W, \Phi) = \{\sigma \mid [\![\, W \,]\!]_{tot,\mathcal{S}}\,(\sigma) \subseteq \Phi\}\ .$$

Weakest (liberal) preconditions, according to this definition, define sets of states. Assertions can be used to define sets of states, too, and by the *Theorem of Definability* there exists a corresponding assertion for every precondition (Apt et al., 2009, Theorem 3.4, p. 87). Therefore, for every weakest (liberal) precondition $wlp(W, [\![\, p \,]\!]_{\mathcal{S}})$ there exists a corresponding *assertion* $wlp(W, p)$ such that $wlp(W, [\![\, p \,]\!]_{\mathcal{S}}) = [\![\, wlp(W, p) \,]\!]_{\mathcal{S}}$. The same is true for $wp(W, [\![\, p \,]\!]_{\mathcal{S}})$.

Apt et al. prove completeness in two steps:

(1) They show that $\vdash_K \{wlp(W, q)\}\, W\, \{q\}$ holds for every wlp (and wp). This proof is done for every workflow statement, or, in case of Apt et al.'s work, While language statement.

(2) They show that whenever a workflow is correct with respect to some p, q, that is, $\models_K \{p\}\, W\, \{q\}$, then it is provably so, $\vdash_K \{p\}\, W\, \{q\}$. This is true because

$\models_K \{p\}\,W\,\{q\}$ if and only if $p \Rightarrow wlp(W,q)$ (Apt et al., 2009, Weakest Liberal Precondition Lemma, p. 87), $\vdash_K \{wlp(W,q)\}\,W\,\{q\}$ (step 1), and the rule of consequence (Apt et al., 2009, p. 90).

As the proof of completeness (2) does not rely on workflow statements directly, but on the general argument of $\vdash_K \{wlp(W,q)\}\,W\,\{q\}$ (and wp), it is sufficient for this thesis to show that argument (1) is indeed true for the new workflow statements, namely service call and `foreach` loop. To this end, we define and prove a Weakest (Liberal) Precondition Lemma to relate the weakest (liberal) preconditions of these statements to assertions. Then, we prove $\vdash_K \{wlp(W,q)\}\,W\,\{q\}$ (and wp) for the new statements. As this is step (1) of the overall proof of completeness, this concludes the proof of Theorem 3.17. For more details on the different steps of this proof and proofs for the standard workflow statements, we refer to Apt et al. 2009, p. 89-91.

Lemma 3.19 (Weakest (Liberal) Preconditions of Workflow Elements)**.** *We use variable, service, and assertion names as before. Then the following statements hold:*

Service Call

$$\begin{aligned} & wlp(u := S(v), q) \\ \Leftrightarrow\ & \forall w : post_S[i := v, o := w] \Rightarrow q[u := w] \end{aligned}$$

and

$$\begin{aligned} & wp(u := S(v), q) \\ \Leftrightarrow\ & pre_S[i := v] \wedge \forall w : (post_S[i := v, o := w] \Rightarrow q[u := w]) \end{aligned}$$

Take

$$\begin{aligned} & wlp(take(a, A), q) \\ \Leftrightarrow\ & \forall b \in A : q[a := b, A := A \setminus \{b\}] \end{aligned}$$

and

$$\begin{aligned} & wp(take(a, A), q) \\ \Leftrightarrow\ & A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \end{aligned}$$

Foreach

$$\begin{aligned} & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, q) \wedge A \neq \emptyset \\ \Rightarrow\ & wlp(take(a, A); W, wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, q)) \\ & \\ & and \\ & \\ & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ , W\ \texttt{od}, q) \wedge A = \emptyset \\ \Rightarrow\ & q \\ & \\ & and \\ & \\ & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ , W\ \texttt{od}, q) \\ \Leftrightarrow\ & wp(\texttt{foreach}\ a \in A\ \texttt{do}\ , W\ \texttt{od}, q) \end{aligned}$$

Proof. We prove the assertions for all three workflow statements separately.

Service Call

As a reminder, the conditions of the transition rule are:

$$\sigma \models_K pre_{Svc}[i := v] \tag{3.10}$$

$$\sigma' \models_K post_{Svc}[i := v, o := u] \tag{3.11}$$

$$\forall x \in Var \setminus \{u\} : \sigma(x) = \sigma'(x) \tag{3.12}$$

For total correctness, by definition,

$$wp(u := S(v), [\![q]\!]_{\mathcal{S}}) = \{\sigma \mid [\![u := S(v)]\!]_{tot,\mathcal{S}}(\sigma) \subseteq [\![q]\!]_{\mathcal{S}}\}$$

and

$$\begin{aligned} & wp(u := S(v), q) \\ \Leftrightarrow\ & pre_S[i := v] \wedge \forall w : post_S[i := v, o := w] \Rightarrow q[u := w]\ . \end{aligned}$$

Let $\sigma \in [\![pre_S[i := v] \wedge \forall w : post_S[i := v, o := w] \Rightarrow q[u := w]]\!]_{\mathcal{S}}$. Then, by the transition rule for service calls, there is $\langle u := S(v), \sigma\rangle \rightarrow_{\mathcal{S}} \langle E, \sigma'\rangle$ with $\sigma' \in [\![q]\!]_{\mathcal{S}}$.

If $\sigma \notin [\![pre_S[i := v] \wedge \forall w : post_S[i := v, o := w] \Rightarrow q[u := w]]\!]_{\mathcal{S}}$, then we distinguish two cases: If $\sigma \notin [\![pre[i := v]_S]\!]_{\mathcal{S}}$, by the transition rule, the workflow blocks, that is, $[\![u := S(v)]\!]_{tot}(\sigma) = \{\perp\}$, and $\{\perp\} \not\subseteq [\![q]\!]_{\mathcal{S}}$. If $\sigma \notin [\![\forall w : post_S[i := v, o := w] \Rightarrow q[u := w]]\!]_{\mathcal{S}}$ (but $\sigma \in [\![pre[i := v]_S]\!]_{\mathcal{S}}$), then there exist σ' and w such that $\sigma' \models_K post_S[i := v, o := w] \wedge \neg q[u := w]$, and $\langle u := S(v), \sigma\rangle \rightarrow_{\mathcal{S}} \langle E, \sigma'\rangle$, but $\sigma' \notin [\![q]\!]_{\mathcal{S}}$.

For partial correctness, by definition,

$$wlp(u := S(v), [\![\, q \,]\!]_{\mathcal{S}}) = \{\sigma \mid [\![\, u := S(v) \,]\!]_{part,\mathcal{S}}\,(\sigma) \subseteq [\![\, q \,]\!]_{\mathcal{S}}\}$$

and

$$\begin{array}{rl} & wlp(u := S(v), q) \\ \Leftrightarrow & \forall w : post_S[i := v, o := w] \Rightarrow q[u := w] \ . \end{array}$$

Let $\sigma \in [\![\, \forall w : post_S[i := v, o := w] \Rightarrow q[u := w] \,]\!]_{\mathcal{S}}$. Then, we distinguish two cases: With $\sigma \in [\![\, pre[i := v]_S \,]\!]_{\mathcal{S}}$, by the transition rule for service calls, there is $\langle u := S(v), \sigma\rangle \to_{\mathcal{S}} \langle E, \sigma'\rangle$ with $\sigma' \in [\![\, q \,]\!]_{\mathcal{S}}$. With $\sigma \notin [\![\, pre[i := v]_S \,]\!]$, by the transition rule, the workflow blocks, that is, $[\![\, u := S(v) \,]\!]_{part,\mathcal{S}}\,(\sigma) = \emptyset$, and $\emptyset \subseteq [\![\, q \,]\!]_{\mathcal{S}}$.

If $\sigma \notin [\![\forall w : post_S[i := v, o := w] \Rightarrow q[u := w] \,]\!]_{\mathcal{S}}$ (but $\sigma \in [\![\, pre[i := v]_S \,]\!]_{\mathcal{S}}$), then there exist σ' and w such that $\sigma' \models_K post_S[i := v, o := w] \wedge \neg q[u := w]$, and $\langle u := S(v), \sigma\rangle \to_{\mathcal{S}} \langle E, \sigma'\rangle$, but $\sigma' \notin [\![\, q \,]\!]_{\mathcal{S}}$.

Take For total correctness, by definition,

$$wp(take(a, A), [\![\, q \,]\!]_{\mathcal{S}}) = \{\sigma \mid [\![\, take(a, A) \,]\!]_{tot,\mathcal{S}}\,(\sigma) \subseteq [\![\, q \,]\!]_{\mathcal{S}}\}$$

and

$$\begin{array}{rl} & wp(take(a, A), q) \\ \Leftrightarrow & A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \ . \end{array}$$

Let $\sigma \in [\![\, A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \,]\!]_{\mathcal{S}}$ and σ' a state such that $\langle take(a, A), \sigma\rangle \to_{\mathcal{S}} \langle E, \sigma'\rangle$. From the transition axiom for *take* we know that whenever there is a b such that

$$\sigma'(b) \in \sigma(A)$$

then

$$\sigma'(A) = \sigma(A) \setminus \sigma'(b) \ ,$$

or, comparing the states directly,

$$\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \quad \sigma' = \sigma[A := A \setminus \{b\}] \ . \tag{3.13}$$

Because

$$\sigma \models_K \big(A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]\big)$$

and therefore also

$$\begin{array}{l} \forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \\ \sigma \models_K \big(A \neq \emptyset \wedge q[a := b, A := A \setminus \{b\}]\big) \end{array}$$

as well as

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) :\\ &\sigma \models_K q[a := b, A := A \setminus \{b\}]\end{aligned}$$

by substitution

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) :\\ &\sigma[A := A \setminus \{b\}] \models_K q[a := b] \ .\end{aligned}$$

Now by (3.13),

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) :\\ &\sigma' \models_K q[a := b] \ ,\end{aligned}$$

and because this holds for every b in the set Var, including the original a,

$$\sigma' \models_K q$$

and therefore $\sigma' \in [\![q]\!]_{\mathcal{S}}$.

For partial correctness, by definition,

$$\begin{aligned}& wlp(take(a, A), q)\\ \Leftrightarrow \quad & \forall b \in A : q[a := b, A := A \setminus \{b\}] \ .\end{aligned}$$

Let $\sigma \in [\![\forall b \in A : q[a := b, A := A \setminus \{b\}]]\!]_{\mathcal{S}}$. Then either $\sigma(A) \neq \emptyset$, and the proof works the same as for total correctness; or $\sigma = \emptyset$, and by the transition rule the workflow blocks, that is,

$$[\![\, take(a, A) \,]\!]_{part,\mathcal{S}} \, (\sigma) = \emptyset \ ,$$

and $\emptyset \subseteq [\![q]\!]_{\mathcal{S}}$.

Foreach, empty set By definition of weakest (liberal) preconditions,

$$wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, [\![q]\!]_{\mathcal{S}})$$

is

$$\{\sigma \mid [\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]_{\mathcal{S}} \, (\sigma) \subseteq [\![q]\!]_{\mathcal{S}}\}$$

and by semantics of `foreach` for an empty set $A = \emptyset$,

$$\subseteq \quad [\![q]\!]_{\mathcal{S}} \ .$$

Therefore

$$wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, [\![\, q \,]\!]_{\mathcal{S}}) \cap [\![\, A = \emptyset \,]\!]_{\mathcal{S}} \subseteq [\![\, q \,]\!]_{\mathcal{S}} \ .$$

By the Theorem of Definability (Apt et al., 2009, Theorem 3.4, p. 87), we can write this as assertion:

$$wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, q) \wedge A = \emptyset \Rightarrow q \ .$$

Foreach, non-empty set By definition of weakest (liberal) preconditions,

$$wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, [\![\, q \,]\!]_{\mathcal{S}})$$

is

$$\{\sigma \mid [\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od} \,]\!]_{\mathcal{S}}(\sigma) \subseteq [\![\, q \,]\!]_{\mathcal{S}}\}$$

and by semantics of `foreach` for a non-empty set $A \neq \emptyset$,

$$\begin{aligned} &\subseteq \quad \{\sigma \mid [\![\, take(a, A); W \,]\!]_{\mathcal{S}}(\sigma) \subseteq [\![\, p \,]\!]_{\mathcal{S}}\} \cap [\![\, A \neq \emptyset \,]\!]_{\mathcal{S}} \\ &\text{s.t. } [\![\, p \,]\!]_{\mathcal{S}} = \{\sigma' \mid [\![\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od} \,]\!]_{\mathcal{S}}(\sigma') \subseteq [\![\, q \,]\!]_{\mathcal{S}}\} \ . \end{aligned}$$

Now, by induction hypothesis,

$$\subseteq \quad \{\sigma \mid [\![\, take(a, A); W \,]\!]_{\mathcal{S}}(\sigma) \subseteq wlp(\texttt{foreach} \ldots, [\![\, q \,]\!]_{\mathcal{S}})\} \ ,$$

which is by definition of weakest (liberal) preconditions

$$\subseteq \quad wlp(take(a, A); W, wlp(\texttt{foreach} \ldots, [\![\, q \,]\!]_{\mathcal{S}})) \ .$$

By the Theorem of Definability (Apt et al., 2009, Theorem 3.4, p. 87), we can write this as assertion:

$$wlp(take(a, A); W, wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, q)) \ .$$

□

Now, we show that $\vdash_K \{wlp(W, q)\}\, W\, \{q\}$ (and $\vdash_K \{wp(W, q)\}\, W\, \{q\}$) for all new preconditions. This concludes the proof of Theorem 3.17. Generally, we need two different proof calculi to prove partial and total correctness of a workflow (Apt et al., 2009, p. 70–71). However, as the proof rules are almost the same, we talk about "the" proof calculus only.

Proof. We prove $\vdash_K \{wlp(W, q)\}\, W\, \{q\}$ (and $\vdash_K \{wp(W, q)\}\, W\, \{q\}$) by induction, using workflows with proof axioms as base cases and workflows with proof rules as induction steps.

Service Call (wp) To show: $\vdash_K \{wp(u := S(v), q)\}\, u := S(v)\, \{q\}$. This is by definition of the weakest precondition for service calls:

$$\vdash_K \{pre_S[i := v] \wedge \forall w : post_S[i := v, o := w] \Rightarrow q[u := w]\}\, u := S(v)\, \{q\} \ ,$$

which is covered by the proof axiom for total correctness for service calls.

Service Call (wlp) To show: $\vdash_K \{wlp(u := S(v), q)\}\, u := S(v)\, \{q\}$. This is by definition of the weakest liberal precondition for service calls:

$$\vdash_K \{\forall w : post_S[i := v, o := w] \Rightarrow q[u := w]\}\, u := S(v)\, \{q\} \ ,$$

which is covered by the proof axiom for partial correctness for service calls.

Take (wp) To show: $\vdash_K \{wp(take(a, A), q)\}\, take(a, A)\, \{q\}$. By definition of weakest precondition for *take*, that is

$$\vdash_K \{A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]\}\, take(a, A)\, \{q\} \ ,$$

which is covered by the proof axiom for total correctness for *take*.

Take (wlp) To show: $\vdash_K \{wlp(take(a, A), q)\}\, take(a, A)\, \{q\}$. By definition of weakest (liberal) precondition for *take*, that is

$$\vdash_K \{\forall b \in A : q[a := b, A := A \setminus \{b\}]\}\, take(a, A)\, \{q\} \ ,$$

which is covered by the proof axiom for partial correctness for *take*.

Foreach To show: $\vdash_K \{wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}, q)\}\, \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}\, \{q\}$. By induction hypothesis, we have

$$\begin{array}{ll}
 & \vdash_K \{wlp(take(a, A); W, wlp(\texttt{for}\ldots, q))\}\, take(a, A); W\, \{wlp(\texttt{for}\ldots, q)\} \\
\Rightarrow & \text{by Lemma 3.19 and rule of consequence} \\
 & \vdash_K \{wlp(\texttt{for}\ldots, q) \wedge A \neq \emptyset\}\, take(a, A); W\, \{wlp(\texttt{for}\ldots, q)\} \\
\Rightarrow & \text{by } \texttt{foreach} \text{ proof rule} \\
 & \vdash_K \{wlp(\texttt{for}\ldots, q)\}\, \texttt{for}\ldots\, \{wlp(\texttt{for}\ldots, q) \wedge A = \emptyset\} \\
\Rightarrow & \text{by Lemma 3.19, rule of consequence and } A = \emptyset \\
 & \vdash_K \{wlp(\texttt{for}\ldots, q)\}\, \texttt{for}\ldots\, \{q\}
\end{array}$$

□

Apt et al. distinguish between proofs for partial and total correctness: For total correctness, termination has to be proven, that is, the alternate proof rule for `while` loops with progress expressions and termination bounds has to be used. In our context, this is not necessary, as `foreach` loops terminate by definition, as their semantics include the reduction of the (finite) iteration set.

Chapter 4

Workflow Templates

Chapter 3 combines classical program verification, a flexible service market and on-the-fly composition, and formalized domain knowledge. We provide a framework to prove correctness of a composition, but we elaborate on *automatic* verification only in Chapter 5, where we will do verification by solving a satisfiability problem. But regardless of whether or not the process of verification can be automated, it has – up to now – the following characteristics: Reasoning always has to include the complete domain knowledge. As ontologies include predicates with no fixed interpretation, the definition of correctness of a composition necessarily argues over all possible logical structures, restricted by its ontology ($\rightarrow$ Definition 3.12, Correctness of Compositions), and therefore all possible interpretations of the predicates of the ontology.

Extrapolating from already existing real-world ontologies, we have to expect formalized domain knowledge in the order of magnitude of up to tens of thousands of concepts and predicates: While the QALLME Tourism ontology[1] only contains a few dozen concepts and roles, the e-commerce taxonomy of *schema.org*[2], which is used by, e.g., Google, contains more than 500 types and 800 properties.[3] The prominently known medical ontology OpenGALEN consists of tens of thousands of concepts and roles.[4] These numbers do neither include additional roles supported natively by OWL (such as sub-classes, sub-roles, transitivity, and similar properties of roles, ...) nor additional rule sets.

From a logical point of view, the number of predicates matters most, as this is the "uninterpreted" part. From a *pragmatic* point of view, the number of concepts matters just as well, as we encode them using predicates. This is necessary to respect the subtype relations from the ontology ($\rightarrow$ Chapters 5 and 7). In comparison to pure program verification, where verification conditions are often considered to be terms of propositional variables, and/or consisting of at most linear algebra operators, this makes

[1]QALLME tourism ontology: `http://qallme.fbk.eu`, retrieved July 21, 2017.

[2]Good Relations ontology project: `http://www.goodrelations-vocabulary.org`, Schema.org e-commerce taxonomy: `http://www.schema.org`, both retrieved July 21, 2017.

[3]`http://schema.org/docs/schemas.html`, retrieved July 21, 2017.

[4]OpenGALEN ontology: `http://www.opengalen.org`, retrieved July 21, 2017.

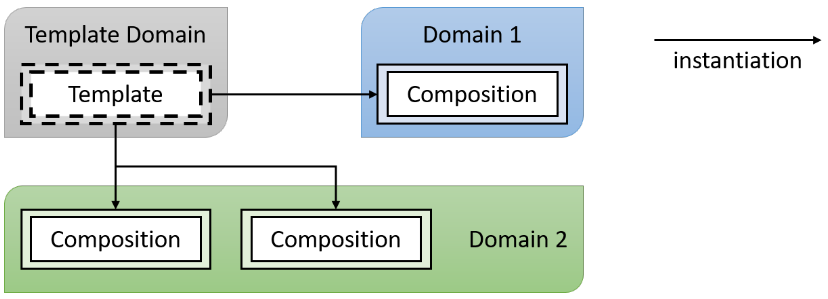

Figure 4.1: Instantiating correct templates replaces the need for composition verification with a check for constraint rule compliance during the instantiation process

a huge difference regarding the state space a solver – or any other state space exploration technique – has to cope with. Additionally, complex domain knowledge paves the way to complex service descriptions, and, consequently, complex service compositions. But in complex domains with extensive use of sub-classing, odds are that repeating *patterns* of service compositions emerge.

There are two main motivations for such patterns, visualized in Figure 4.1: On the one hand, the task may be as abstract as a generic sorting service, or a service providing a learning algorithm. In this case, the composition – or workflow, or algorithm – itself is even independent of the domain, but it can be used with concrete concepts of the domain. On the other hand, a composition may be quite complex and comprise the expertise of domain experts. However, while the overall workflow is very (domain) specific, the exact services which are used in the composition are not necessarily predetermined. Instead, it may be sufficient to use “a sorting service” or a “payment processing service”, without going into the details.

Why is this relevant? In an on-the-fly context, the *requirements* of a service composition, that is, the description of the service composition that should be delivered, are quite specific. A user (end user or business user) would not request a “generic sorting service” but a “service to sort data XY according to Z”. Therefore, even an abstract composition is typically matched by a specific request. This leads to the idea of *template compositions* (templates for short). In several programming paradigms like object oriented programming (OO) and component-based software engineering (CBSE) the idea of templates is quite prevalent. In C++, templates are an explicit programming construct to create blueprints for, e.g., object-oriented classes. Java uses *generics* to introduce type variables, which can be used to implement template-style classes and methods. Template ideas are supported either directly as language construct to create class structures, or indirectly by using inheritance and subtyping, and are therefore built-in into every language with a polymorphic typing system. Section 4.1 gives an introduction to template approaches in the literature, both in programming and in process modeling.

In summary, we can conjecture the following: (a) The complexity of verification depends not only on the complexity of the composition, but also on the size (and complexity) of the knowledge base; (b) While there may exist patterns for abstract tasks, concrete requirements are always specific (instead of generic).

This enables us to make an important simplification in the process of verifying service compositions: With *abstract service compositions* or *templates*, we define the abstract part of the recurring task. We verify it at an abstract level, before we know any specific requirements (though we can formalize abstract requirements, of course). To be able to do that, we collect all side-conditions (or *constraints*) which are necessary to prove the correctness of the composition. With a verified template, and a concrete requirements present, we can *instantiate* the template with concrete services in the current domain. While instantiation itself is nothing new, our approach allows for instantiations which are correct by construction, as long as the collected constraints are not violated. The key advantage is that the instantiation – a service composition with the full expressiveness of the knowledge base of the current domain – does not need to be verified *again*. Instead, a check of the constraints takes place within the ontology itself, which is a task that can typically be reduced to a concept-inclusion check of specialized reasoners (Schmidt-Schauß and Smolka, 1991, and, e.g., Tsarkov and Horrocks, 2006).

We can use this technique not only to model abstract tasks within a domain, but also domain-independently. Then, we define an abstract domain which only includes the concepts and rules absolutely necessary to specify and verify the template. After that, an instantiation to a compatible target domain leverages the reuse of the template, again with instantiations which are correct by construction.

The remainder of this chapter, after giving an introduction to templates in general and templates in verification (Section 4.1), supplements the syntax and semantics of workflows to be able to deal with *service placeholders*, and therefore defining a *template* language (Section 4.2). Section 4.3 defines parameterized correctness for templates, while Section 4.4 updates the proof calculus, where necessary. Section 4.5 formalizes the instantiation of templates and elaborates on the constraints, which guarantee a correctness-by-construction of the instantiated service compositions.

The core elements of this chapter, especially the definitions leading to Theorem 4.24, the theorem itself and its proof have been published in Walther and Wehrheim (2015) and its journal version Walther and Wehrheim (2016).

4.1 Related Work

Workflow *templates*, where certain parameters can be instantiated with actual values into an actual workflow, are common on the level of programming languages. There, they take the form of, e.g., C++ Templates or Java Generics, where *type variables* are used in code, which are instantiated with concrete types whenever a new object of a parameterized class is created. Aside from explicit language constructs for template elements, type hierarchies allow for the use of abstract programs and instantiations with concrete (sub-) types as well.

Design patterns for common programming problems are the most prominent use of this principle, not only in programming (Gamma et al., 1994), but also in workflow modeling (van der Aalst and ter Hofstede, 2012). These patterns, as well as function pointers or the introduction of lambda expressions to (imperative) programming languages, allow for rather flexible template programs, where not only types, but also function calls can be substituted by parameters.

Slightly more abstract, but related, is the topic of *code variability*. Here, function call placeholders (or contracts) are not only substituted by concrete versions, but they are also described using pre- and postconditions (e.g., Soleimanifard and Gurov, 2015). Product line analysis revolves around the same topics, though the ultimate goal is not template specification but configuration analysis (Apel et al., 2013). Similarly, delta-oriented programming verifies a “core” program (or template) and different program refinements (“deltas”), combining the results (Hähnle and Schaefer, 2012).

Sirin, Parsia, and Hendler propose to use the existing service composition language OWL-S to model templates. To this end, they introduce a new atomic workflow step to represent an *abstract process*, in contrast to an *atomic process*. An OWL-S atomic process is the equivalent of a service call, and it is used in the process description. An atomic process itself is linked to a WSDL description and annotated with pre- and postconditions. Therefore the *abstract process* is effectively a *service placeholder*. Using OWL-S process descriptions, it is possible to define workflow templates mixing both concrete services, or atomic processes, and service placeholders, or abstract processes (Sirin et al., 2005).

Gil, Groth, Ratnakar, and Fritz propose workflow templates in the context of executing computational experiments. While their approach is based on command line applications as “services”, the principle can be adapted to Web services. However, their focus is on the treatment of data set collections and workflow repetitions with changing parameter sets. They build upon ontologies (using OWL) to model a custom workflow template language, treating complex input and output data as variables and using custom postprocessing to translate a template into a workflow instance. As an additional restriction, they do not use loops, but only sequences, conditionals and parallel execution (Gil et al., 2009, 2011).

Fleuren, Götze, and Müller follow a similar approach, without using ontologies but creating a custom specification language. However, their focus is on instantiating workflows with a variable number of parallel sub-processes (or services), thus their workflow languages revolves around parameterizing parallel execution constructs (Fleuren et al., 2014).

Summarizing, for template approaches it is also quite common to cover two of the three core topics of semantic service descriptions, workflow semantics, and knowledge formalization, but combining all three with an underlying theory is an open issue.

4.2 Syntax and Semantics

A service composition *template* is, at a first glance, just another service composition, but using “placeholders” instead of real service descriptions. However, this is just one of two differences. From a syntactical point of view, this is indeed correct: A template, by its very idea, makes use of *service placeholders*, because the real services will only be determined in some kind of *instantiation*. All control from structures remain the same. But instead of introducing a new workflow statement to deal with service placeholders, we will define service placeholders as a specific type of regular service descriptions. This way, the workflow language syntax does not need to be changed.

Definition 4.1 (Service Placeholder)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, with $\mathcal{T}_K$ the set of types associated with this domain. Let $Sp^{Sig} = Sp : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_n$ be the signature of a service Sp. Then

$$Sp^{Desc} = (Sp^{Sig}, I_{Sp}, O_{Sp}, pre_{Sp}, post_{Sp})$$

denotes the *service placeholder Sp*, with

- the service placeholder’s signature Sp^{Sig},
- input variables I_{Sp} with $|I_{Sp}| = k$,
- output variables O_{Sp} with $|O_{Sp}| = (n - k)$,
- a predicate pre_{Sp} with $free(pre_{Sp}) \subseteq I_{Sp}$,
- a predicate $post_{Sp}$ with $free(post_{Sp}) \subseteq (I_{Sp} \cup O_{Sp})$,
- and $I_{Sp} \cap O_{Sp} = \emptyset$.

We assume I_{Sp} and O_{Sp} to be ordered such that variable names uniquely correspond to a position in Sp^{Sig}. We denote the set of all services placeholders of a domain with knowledge base K as $\mathcal{SP}_K$.

Pre- and postconditions of service placeholders are predicates, in contrast to the pre- and postconditions of services, which are terms. These predicates are not part of the ontology, and therefore cannot be part of terms (as Φ_K is defined based on the ontology), but as they will be an important part of workflow templates, we supplement the set of formulas by these placeholder predicates.

Definition 4.2 (Formulas with Placeholder Predicates)**.** For a knowledge base K and a set of formulas Φ_K, let

$$\Phi_K^{sp} = \Phi_K \cup \{pre_{sp}, post_{sp} \mid sp \in \mathcal{SP}_K\} \ .$$

The following service placeholder denotes a service which retrieves additional data for a given input.

Example 4.1 (Service Placeholder Example)**.** *For readability, we specify the service placeholder in table structure.*

Placeholder	**Inputs**	**Outputs**
Acquire	x : *InputData*	y : *OutputData*

In contrast to services, there are not pre- and postconditions, as they are represented automatically by the predicates $pre_{Acquire}$ *and* $post_{Acquire}$*.*

For services, pre- and postconditions are terms built from the vocabulary of the knowledge base, while for placeholders, they are additional predicates. While the syntax of workflows can be directly reused, the introduction of service placeholders has a major impact on the semantics.

Please note, that the definition of service *placeholders* and actual *services* differ only in the pre- and postcondition. The behavior of an (actual) service is determined by its pre- and postconditions, that is, by logical formulas that capture the characteristics of the service's input and output data. A placeholder is different: Here, pre- and postcondition are predicates. In other words: A service placeholder may accept or generate basically arbitrary data, as the pre- and postcondition predicates are not derived from the domain ontology, and they are therefore totally unrestricted when it comes to their interpretation.

This circumstance leads us to introduce two additional elements: At first, it is of no use to define correctness for a template with just placeholders. Therefore, we introduce a *mapping* π *from placeholders to services*, which will become useful in the definition of correctness as well as the definition of template instantiations in Section 4.5. At second, we introduce *workflow templates* as a variant of workflows, and define their semantics using the service mapping π. Using π, we can map a template to every possible instantiation. Our only assumption about π is that it respects the signature of the service placeholders.

Definition 4.3 (Service Mapping)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, with $\mathcal{T}_K$ the set of types associated with this domain. Then

$$\pi : \mathcal{SP}_K \to \mathcal{SVC}_K$$

denotes the *service mapping* which maps service placeholders to actual services within the domain of K, with the property that it respects signatures, that is,

$$\begin{aligned}\text{if } \pi(sp) = svc \text{ and } sp : T_1 \times \cdots \times T_i \to T_{i+1} \times \cdots \times T_n \\ \text{then } svc : T'_1 \times \cdots \times T'_i \to T'_{i+1} \times \cdots \times T'_n \\ \text{with } T_k \sqsubseteq T'_k \text{ for } 1 \leq k \leq i \\ \text{and } T'_k \sqsubseteq T_k \text{ for } i < k \leq n \ .\end{aligned}$$

The service mapping respects the Liskov substitution principle (Liskov and Wing, 1994). Section 4.5.2 elaborates this topic in more details, when intra-domain service mapping is extended to mapping between two different domains.

The *syntax* of workflows does not change for workflow templates. The only difference is that service calls do not necessarily use the names of *services*, but also of *service placeholders*.

The service mapping maps descriptions of service placeholders to descriptions of actual services, but as we reason about formulas we are additionally interested in replacing pre- and postconditions of service placeholders (in a formula) with their counterparts of the mapped services. Therefore, we lift the service mapping from service descriptions to formulas.

Definition 4.4 (Placeholder Predicate Replacement)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. We lift π from $\mathcal{SP}_K \to \mathcal{SVC}_K$ to $\Phi_K^{sp} \to \Phi_K$ such that for $\varphi \in \Phi_K^{sp}$

$$\pi(\varphi) = \begin{cases} pre_{\pi(sp)} & \text{if } \varphi = pre_{sp} \\ post_{\pi(sp)} & \text{if } \varphi = post_{sp} \\ P(x,y) & \text{if } \varphi = P(x,y) \text{ and } P \in \mathcal{P}_K \\ F(x,y) & \text{if } \varphi = F(x,y) \text{ and } F \in \mathcal{F}_K \\ \neg\pi(\varphi_1) & \text{if } \varphi = \neg\varphi_1 \\ \pi(\varphi_1) \vee \pi(\varphi_2) & \text{if } \varphi = \varphi_1 \vee \varphi_2 \ . \end{cases}$$

Having a connection of service placeholders and services, we can define a workflow template.

Definition 4.5 (Workflow Template)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. The following rules define the syntax of a *workflow template* WT:

$$
\begin{aligned}
WT ::= & \; [l] \; \mathit{skip} \\
\mid & \; [l] \; u := t \\
\mid & \; WT_1; WT_2 \\
\mid & \; [l] \; (u_{j+1}, \ldots, u_k) := S(i_1, \ldots, i_j) \\
\mid & \; [l] \; \texttt{if } B \texttt{ then } WT_1 \texttt{ else } WT_2 \texttt{ fi} \\
\mid & \; [l] \; \texttt{while } B \texttt{ do } WT_1 \texttt{ od} \\
\mid & \; [l] \; \texttt{foreach } a \in A \texttt{ do } WT_1 \texttt{ od}
\end{aligned}
$$

with $u, v, a, A \in \mathit{Var}$, $t \in \Phi_K$, $\mathit{type}(t) = \mathit{type}(u)$, $\mathit{type}(A) = \textbf{set } T, \mathit{type}(a) = T, T \in \mathcal{T}_K$, $B \in \Phi_K$, and $S \in \mathcal{SVC}_K \cup \mathcal{SP}_K$. Additionally, we define exactly one final label $[\mathit{end}]$ for a workflow template WT.

The difference to workflow definitions ($\rightarrow$ Definition 3.3, Workflow) is that names for service calls can not only refer to real services, but also to service placeholders. The *semantics* of workflow templates, in the presence of service placeholders, now depends on the actual services which replace the placeholders. Therefore, we change the workflow semantics from Definition 3.8 by adding the mapping function as an additional parameter. This implies, of course, that the *exact* semantics is not determined until a concrete π is given. Additionally, we parameterize transitions with π and use it in the application condition of some of the rules.

Definition 4.6 (Termination, Blocking, and Divergence of Workflow Templates)**.** A *transition sequence for workflow templates* is a finite or infinite sequence of configurations $\langle WT_1, \sigma_1 \rangle \rightarrow^{\pi}_{\mathcal{S}} \cdots \rightarrow^{\pi}_{\mathcal{S}} \langle WT_i, \sigma_i \rangle \ldots$ A *computation* of a workflow template WT starting in σ is a transition sequence which either (a) cannot be extended, or (b) is infinite. A computation *terminates*, if it cannot be extended and ends with $\langle E, \tau \rangle$, that is, the empty workflow. A computation *blocks*, if it cannot be extended, but ends with $\langle WT', \tau \rangle$ and $WT' \not\equiv E$. A computation *diverges*, if it is infinite.

As for standard workflows, we use $\rightarrow^{\pi*}_{\mathcal{S}}$ to denote the transitive closure of $\rightarrow^{\pi}_{\mathcal{S}}$.

This implies, that it depends on the actual service placeholder mapping whether or not a transition rule (or axiom) is valid for a given workflow.

Definition 4.7 (Transition Axioms and Rules for Templates)**.** For a knowledge base K, let $u, v, a, A \in \mathit{Var}$, $t \in \Phi_K$, $\mathit{type}(t) = \mathit{type}(u)$, $\mathit{type}(A) = \textbf{set } T, \mathit{type}(a) = T, T \in \mathcal{T}_K$, $B \in \Phi_K$, and $\mathit{Svc} \in \mathcal{SVC}_K \cup \mathcal{SP}_K$. We define the operational semantics for a parameterized workflow template WT and state σ with a service mapping π inductively as follows:

Skip	$\langle \mathit{skip}, \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle$
Assignment	$\langle u := t, \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma[u := \sigma(t)] \rangle$
Take	$\langle \mathit{take}(a, A), \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma' \rangle$
if		$\sigma'(a) \in \sigma(A), \sigma'(A) = \sigma(A) \setminus \{\sigma'(a)\},$
and		$\sigma(x) = \sigma'(x)$ for $x \neq A, x \neq a$
Sequential comp.	$\dfrac{\langle WT_1, \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle WT_2, \tau \rangle}{\langle WT_1; WT, \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle WT_2; WT, \tau \rangle}$	
Cond. (then)	$\langle \textbf{if } B \textbf{ then } WT_1 \textbf{ else } WT_2 \textbf{ fi}, \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle WT_1, \sigma \rangle$	
if		$\sigma \models_{\mathcal{S}} \pi(B)$
Cond. (else)	$\langle \textbf{if } B \textbf{ then } WT_1 \textbf{ else } WT_2 \textbf{ fi}, \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle WT_2, \sigma \rangle$	
if		$\sigma \models_{\mathcal{S}} \neg\pi(B)$
While (loop)	$\langle \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle WT; \textbf{while } B \textbf{ do } WT \textbf{ od}, \sigma \rangle$
if		$\sigma \models_{\mathcal{S}} \pi(B)$
While (end)	$\langle \textbf{while } B \textbf{ do } WT \textbf{ od}, \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle$
if		$\sigma \models_{\mathcal{S}} \neg\pi(B)$
Foreach (loop)	$\langle \textbf{foreach } a \in A \textbf{ do } WT \textbf{ od}, \sigma \rangle$	
	$\rightarrow^{\pi}_{\mathcal{S}} \langle \mathit{take}(a, A); WT; \textbf{foreach } a \in A \textbf{ do } WT \textbf{ od}, \sigma \rangle$	
if		$\sigma \models_{\mathcal{S}} A \neq \emptyset$
Foreach (end)	$\langle \textbf{foreach } a \in A \textbf{ do } WT \textbf{ od}, \sigma \rangle$	$\rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle$
if		$\sigma \models_{\mathcal{S}} A = \emptyset$
Service call	$\langle u := \mathit{Svc}(v), \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma' \rangle$	
if		$\sigma \models_{\mathcal{S}} \pi(\mathit{pre}_{\mathit{Svc}}[i := v]),$
		$\sigma' \models_{\mathcal{S}} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u]),$
and		$\forall x \neq u : \sigma'(x) = \sigma(x)$

The transition rule for service calls uses the pre- and postcondition from S^{Sig} of the service S. As this is a *template* semantics, "services" include "service placeholders". Therefore, if S refers to an actual service, *pre* and *post* are terms from the service's signature. If S refers to a service placeholder, *pre* and *post* are *predicates* from the service *placeholder's* signature. In the second case, the service mapping π replaces them with the terms from the actual service which belongs to the placeholder.

With a workflow semantics which is parameterized with a placeholder/service mapping we can already formalize service composition templates, under one important condition: Their description stems from a given domain (and is backed by a formalized knowledge base), and every logical formula contains only vocabulary from the ontology and the service placeholder predicates (their pre- and postcondition predicates).

For the *correctness* of templates, it is not possible to show correctness for all possible instantiations directly. Assume two different services S_1 and S_2 accept the same input (have the same precondition), but deliver the opposite output ($\mathit{post}_{S_1} = \neg\mathit{post}_{S_2}$). A template is instantiated using service S_1, and the resulting composition is correct. Assume the correctness depends on the postcondition of S_1. Then, another instantiation of the same template, but with service S_2 instead of S_1 cannot be correct. Therefore, correctness proofs directed at "all possible instantiations", that is, using service placeholders as real services, will fail. To address this issue, and to make the idea of templates feasible again, we introduce one important element to templates, in comparison to pure service compositions: Constraints.

Most likely, service placeholders in a workflow are not as arbitrary as they may appear on a very abstract level. Every service placeholder is part of the workflow *on purpose*, so its behavior – therefore, its postcondition – is required to reach the goal of the overall composition. Using just a generic predicate to represent a placeholder's postcondition, this purpose is only very loosely fixed, namely only in terms of a signature. Predicates in a knowledge base, on the contrary, are limited in their final interpretations by their context, given by ontology-specific properties (as transitivity), inheritance, or rule compliance. In a template, we add such constraints artificially. We allow for *constraint rules*, which relate predicates used in the workflow. Of a special importance are placeholder pre- and postconditions. Constraint rules are the tools to restrict their possible interpretation in exactly such ways as the template designer has in mind.

Example 4.2 (Constraint Rule for Service Placeholders)**.** *The service placeholder from Example 4.1 accepts some input and delivers the corresponding output.*

Placeholder	**Inputs**	**Outputs**
Acquire	x : *Element*	y : *Value*

Assume a relation hasValue : Element×Value *in the template ontology. We want the placeholder to acquire only outputs which correspond to the input, and use the constraint rule*

$$\forall x, y : pre_{Acquire}(x) \land post_{Acquire}(x, y) \Rightarrow hasValue(x, y)$$

to formalize it.

As the example makes clear, constraint rules can be combined with postcondition predicates of service placeholders, and therefore restrict possible interpretations. This will make a notion of correctness for *all possible services* for a placeholder feasible again, *as long as the constraint rules are respected.*

We now define a service composition template, before the following sections go into the details of correctness.

Definition 4.8 (Service Composition Template)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. A *service composition template* (or *template* for short) (Sct^{Desc}, WT, CR) consists of a service description $Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct})$, a workflow template WT, and a set of constraint rules $CR \subseteq \Phi_K^{sp}$.

As Φ_K^{sp} also contains the pre- and postcondition predicates of service placeholders, constraint formulas can combine placeholders with actual ontological knowledge.

The following example represents a generic filtering service, which accepts a set of input data and returns the subset that matches an arbitrary filter criterion, defined by a service placeholder. For this example, we create an ontology which contains just the concepts and roles we need to model the template.

Example 4.3 (Abstract Filter Ontology)**.** *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ define the template's ontology. The role "is target element" describes the property of the set elements which shall be used as a filter. The role "is target value" describes the predicate which provides the actual filter criterion.*

$$\begin{aligned}
C_T &= \{\textsc{Element}, \textsc{Value}\} \\
P_T &= \{\textsc{hasValue} : \textsc{Element} \times \textsc{Value}\ , \\
&\qquad \textsc{isTargetElement} : \textsc{Element} \times \textsc{Bool}\ , \\
&\qquad \textsc{isTargetValue} : \textsc{Value} \times \textsc{Bool}\} \\
\sqsubseteq_T &= \{\} \\
R_T &= \{\mathit{functional}(\textsc{hasValue}), \mathit{functional}(\textsc{isTargetElement}), \\
&\qquad \mathit{functional}(\textsc{isTargetValue})\}
\end{aligned}$$

The predicates are functional to guarantee exactly one pair for any given element and output value.

The template makes use of this ontology.

Example 4.4 (The FILTER Service Composition Template)**.** *The* FILTER *template accepts a set of input data and delivers the subset that matches a filter criterion. This criterion is not directly linked to the input data, but to another type, which has to be fetched first. Figure 4.2 shows the workflow.*

4.3 Partial and Total Correctness

The semantics of workflows and workflow templates differ in the use of the service mapping as a parameter. Consequently, we have to adapt the partial and total correctness semantics of workflow templates. Basically, both semantics are still a mapping from a set of starting states to a set of possible final states, according to the transitive closure of the transition relation $\rightarrow_{\mathcal{S}}^{\pi *}$. Their definitions do not significantly differ from their workflow counterparts, with the exception that the transition between configurations is not only parameterized with a logical structure, but also with a service mapping, as this mapping is part of the template semantics.

Definition 4.9 (Partial Correctness Semantics of Workflow Templates)**.** Let WT be a workflow template and $\sigma \in \Sigma$ be a state. We define the *partial correctness semantics of workflow templates* as a mapping

$$[\![\ WT\ ]\!]^{\pi}_{part,\mathcal{S}} : \Sigma \to 2^{\Sigma}$$

Template	FILTER
Inputs	A : **set** Element
Outputs	B : **set** Element
Precond.	$\forall a \in A : \mathit{pre}_{\mathit{Acquire}}(a)$
Postcond.	$B = \{b \in A \mid \mathit{isTargetElement}(b)\}$
Constraints	$\forall x, y : \mathit{post}_{\mathit{Acquire}}(x, y) \wedge \mathit{isTargetValue}(y) \Rightarrow \mathit{isTargetElement}(x)$ $\forall x, y : \mathit{post}_{\mathit{Acquire}}(x, y) \wedge \neg \mathit{isTargetValue}(y) \Rightarrow \neg \mathit{isTargetElement}(x)$

Workflow

```
$Z := A$;
$B := \emptyset$ ;
foreach $z \in Z$ do
    $y := \mathit{Acquire}(z)$ ;
    if $\mathit{isTargetValue}(y)$ then
        $B := B \cup \{z\}$
    else
        skip
    fi
od
```

FIGURE 4.2: The FILTER template to filter a set

with

$$[\![\ WT\ ]\!]^{\pi}_{part,\mathcal{S}}(\sigma) = \{\tau \mid \langle WT, \sigma \rangle \rightarrow^{\pi *}_{\mathcal{S}} \langle E, \tau \rangle\} \ .$$

Definition 4.10 (Total Correctness Semantics of Workflow Templates)**.** Let WT be a workflow template and $\sigma \in \Sigma$ be a state. We define the *total correctness semantics of workflow templates* as a mapping

$$[\![\ WT\ ]\!]^{\pi}_{tot,\mathcal{S}} : \Sigma \rightarrow 2^{\Sigma \cup \{\bot\}}$$

with

$$[\![\ WT\ ]\!]^{\pi}_{tot,\mathcal{S}}(\sigma) = [\![\ WT\ ]\!]^{\pi}_{part,\mathcal{S}}(\sigma) \cup \{\bot \mid WT \text{ diverges or blocks}\} \ .$$

As with semantics of workflows, we write $[\![\ WT\ ]\!]^{\pi}_{\mathcal{S}}$ to denote both partial and total correctness semantics, whenever appropriate.

Correctness of service compositions is defined only over logical structures which adhere to the rules of the underlying knowledge base, that is, only those structures that allow for a tautological interpretation of R for a given $K = (C, P, \sqsubseteq, R)$ ($\rightarrow$ Definition 3.12, Correctness of Compositions). For service composition *templates* with the presence of service placeholders and therefore potentially unrestricted pre- and postcondition predicates, we introduced *constraint rules* to add intended relationships between the behavior of a service and the overall outcome of the composition. When it comes to correctness, from a designer's perspective, these constraints have to be treated as *facts*, that is, they have the same role as the rules encoded in the knowledge base of the domain.

In other words, in a definition of correctness, we only consider logical structures which adhere not only to the knowledge base, but to the constraint rules of the template as well. Additionally, correctness can only by defined over all possible instantiations π.

Definition 4.11 (Correctness of Composition Templates)**.** Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain. Let (Sct^{Desc}, WT, CR) be a composition template from that domain with a description $Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct})$. The composition template is *correct with respect to its pre- and postcondition*, if and only if

$$\forall\pi, \forall\mathcal{S} \text{ with } \mathcal{S} \models K \text{ and } \mathcal{S} \models CR : [\![\, WT \,]\!]^{\pi}_{\mathcal{S}}([\![\, \pi(pre_{Sct}) \,]\!]_{\mathcal{S}}) \subseteq [\![\, \pi(post_{Sct}) \,]\!]_{\mathcal{S}} \ .$$

Correctness is always partial or total correctness (depending on the semantics). Following Hoare-style notation, we write $\models^{CR}_{K} \{pre_{Sc}\}\ WT\ \{post_{Sc}\}$ to denote that a workflow template WT is correct with respect to this definition under K and the constraint rules CR.

Again, for easier reading, we will sometimes use the name of the composition template instead of the workflow template, that is, instead of

$$\begin{array}{l} \models^{CR}_{K} \{pre_{Sct}\}\ WT\ \{post_{Sct}\} \\ \text{for } (Sct^{Desc}, W) \text{ with } Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct}) \end{array}$$

we just write

$$\models^{CR}_{K} \{pre_{Sct}\}\ Sct\ \{post_{Sct}\} \ .$$

4.4 Proof Calculus

The semantics of service composition templates (or, more precisely, workflow templates) differs from the semantics of service compositions (or workflows) only slightly. While the syntax is basically the same (service placeholders are additionally allowed for service calls, and their pre/post predicates can be used in terms), their semantics are the same as the semantics of workflows, but parameterized with a service mapping. However, technically it is a different semantics, and to provide a consequent logical foundation, we have to adapt our proof rules from Section 3.4. We also have to revise the proofs of soundness and correctness slightly.

4.4.1 Axioms and Rules

The proof calculus for correctness of workflows is defined on a syntactical level. Workflows and workflow templates have the same syntax, with two exceptions: For templates, service placeholder names can be used for service calls, and service placeholder pre- and postcondition predicates can be used in logical expressions (e.g., in a conditional statement). Therefore, the proof axioms and rules (1)–(8) do not need to change, as they work completely on syntax (→ Definition 3.13, Parameterized Proof Calculus for Workflows). However, the hypothesis of rule (9), the rule of consequence, uses the validity of implications for a given domain knowledge, and, consequently, under all possible logical structures which are valid for this domain knowledge.

Correctness of a template is not defined over all logical structures (for a given domain knowledge K), but restricted to structures which comply with the constraint rules of the template. To reflect this, we adapt the definition of proof rule (9) as well as the definition of restricted validity (→ Definition 3.14, Restricted Validity) to include the constraints CR.

Definition 4.12 (Parameterized Proof Calculus for Workflow Templates)**.** For a knowledge base K, let $p, q, t, pre,\ post \in \Phi_K^{sp}$, $i, o, u, v, w, a, A \in Var$, $type(t) = type(u)$, $type(A) = \textbf{set}\ T, type(a) = T, T \in \mathcal{T}_K$, $B \in \Phi_K^{sp}$, $Svc \in \mathcal{SVC}_K \cup \mathcal{SP}_K$, and W, W_1, W_2 and WT workflow templates and CR the constraint rules of WT. Then the axioms and rules of Definition 3.13 (Parameterized Proof Calculus for Workflows), with rule (9) replaced by the following rule, define a *parameterized proof calculus for workflow templates.*

(9) Consequence
$$\frac{\models_K^{CR} (p \Rightarrow r), \{r\}\, WT \{s\}, \models_K^{CR} (s \Rightarrow q)}{\{p\}\, WT \{q\}}$$

Definition 4.13 (Restricted Validity under Constraint Rules)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base, $(Sp^{Desc}, W) \in \mathcal{SP}_K$, and let $p \in \Phi_K^{sp}$. We call p as *valid under K and CR*, if it is valid (a tautology) for every logical structure $\mathcal{S}$ which adheres to the knowledge base and the constraint rules CR of Sp, that is, the following holds:

$$\forall \mathcal{S} : \big(\mathcal{S} \models R\big) \wedge \big(\mathcal{S} \models CR\big) \ \Rightarrow\ \mathcal{S} \models p\ .$$

In short, we write $\models_K^{CR} p$.

Just as for workflows, we denote an existing proof with $\vdash_K^{CR}$.

Definition 4.14 (Provable Properties)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base, and (Sp^{Desc}, WT) a service composition template. Then

$$\vdash_K^{CR} \{p\}\ WT\ \{q\}$$

denotes that there exists a proof outline for WT under K and CR.

4.4.2 Soundness and Completeness

As an analogon to Theorem 3.16, we state soundness of the proof calculus for templates: Whenever we prove a template as correct using the calculus, it is indeed correct by the semantics-based definition of correctness.

Theorem 4.15 (Soundness of Proof Calculus for Templates). *Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, and let (Sct^{Desc}, WT, CR) with $Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct})$ be a service composition template. If Sct can be proven to be correct using the proof calculus (→ Definition 4.12), then it is correct according to the definition of correctness of service composition templates (→ Definition 4.11):*

$$\vdash_K^{CR} \{pre_{Sct}\}\ WT\ \{post_{Sct}\} \quad \Rightarrow \quad \models_K^{CR} \{pre_{Sct}\}\ WT\ \{post_{Sct}\}$$

Consequently, we also adapt the completeness theorem: Whenever a template is semantically correct, there exists a proof in the calculus.

Theorem 4.16 (Completeness of Proof Calculus). *Let $K = (C, P, \sqsubseteq, R)$ be the knowledge base of the current domain, and let (Sct^{Desc}, WT, CR) with $Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct})$ be a service composition template. If Sct is correct according to the definition of correctness of service composition templates (→ Definition 4.11), then correctness can be proven using the proof calculus (→ Definition 4.12):*

$$\models_K^{CR} \{pre_{Sct}\}\ WT\ \{post_{Sct}\} \quad \Rightarrow \quad \vdash_K^{CR} \{pre_{Sct}\}\ WT\ \{post_{Sct}\}$$

The proofs of both theorems do not differ structurally from the proofs of the corresponding theorems 3.16 and 3.17 for workflows. Instead of workflows, they work on workflow templates and, consequently, using the semantics of workflow templates parameterized with a service mapping π.

The fundamental difference to the corresponding proofs on workflows is the impact of the service mapping parameter: While correctness of a workflow is always proved or disproved for exactly one workflow, correctness of a workflow *template* is always shown for *all possible instantiations* of that template. In other words, the proofs have to hold for all mappings π which satisfy the ontology and the constraint rules *CR* of the workflow template.

4.4.2.1 Proof of Soundness

As in the proof of Theorem 3.16 (Soundness of Proof Calculus), we prove soundness of the calculus by induction. Again, following Apt et al. (2009) and extending their calculus with axioms and rules for service calls, the *take* and `foreach` statements, and the modified rule of consequence, we will only treat these here. To make the following proofs more readable, we use service calls with exactly one input and one output.

Proof of Soundness of Proof Calculus for Templates. Let K be a knowledge base, $Svc \in \mathcal{SVC}_K \cup \mathcal{SP}_K$, $A, a, b, u, v, w \in Var$, A of a set type, and $q, t \in \Phi_K$. Let $\pi : \mathcal{SVC}_K \rightarrow \mathcal{SP}_K$ be a service mapping.

Service Call

For total correctness, we need to show that

$$\begin{aligned}&[\![\, u := \mathit{Svc}(v) \,]\!]^{\pi}_{tot,\mathcal{S}} \,(\, [\![\, \mathit{pre}_{\mathit{Svc}}[i := v] \wedge \forall w \text{ s.t. } \mathit{post}_{\mathit{Svc}}[i := v, o := w] : q[u := w] \,]\!]_{\mathcal{S}} \,) \\ &\qquad \subseteq [\![\, q \,]\!]_{\mathcal{S}}\end{aligned}$$

holds. As a reminder, the conditions of the transition rule are:

$$\sigma \models^{CR}_{K} \pi(\mathit{pre}_{\mathit{Svc}}[i := v]) \tag{4.2}$$

$$\sigma' \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u]) \tag{4.3}$$

$$\forall x \in \mathit{Var} \setminus \{u\} : \sigma(x) = \sigma'(x) \tag{4.4}$$

For easier reading, let

$$p := \quad \mathit{pre}_{\mathit{Svc}}[i := v] \wedge \forall w \text{ s.t. } \mathit{post}_{\mathit{Svc}}[i := v, o := w] : q[u := w] \ .$$

Now, let π be a service mapping and σ, σ' be states such that $\sigma \models^{CR}_{K} \pi(p)$ and $\langle u := \mathit{Svc}(v), \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma' \rangle$. By (4.3), we know that

$$\sigma' \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u]) \tag{4.5}$$

and by (4.4) that the states are equal except for the valuation of u:

$$\exists w : \sigma' = \sigma[u := w] \ . \tag{4.6}$$

The valuation of u is not arbitrary because of (4.3), therefore we choose w such that

$$\sigma'(w) = \sigma'(u) \tag{4.7}$$

and therefore

$$\begin{aligned} & \sigma' \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u][u := w]) \\ \Leftrightarrow \quad & \sigma' \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := w]) \ . \end{aligned}$$

Now, (4.6) and (4.5) lead to

$$\sigma[u := w] \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u])$$

and

$$\sigma \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u][u := w]) \ .$$

By $\sigma \in [\![\, p \,]\!]$ and our knowledge about w in (4.7),

$$\sigma \models^{CR}_{K} \pi(\mathit{post}_{\mathit{Svc}}[i := v, o := u][u := w]) \wedge q[u := w]$$

and again

$$\sigma[u := w] \models^{CR}_{K} \pi(post_{Svc}[i := v, o := u]) \wedge q \ ,$$

therefore by (4.6),

$$\sigma' \models^{CR}_{K} \pi(post_{Svc}[i := v, o := u]) \wedge q$$

Then also

$$\sigma' \models^{CR}_{K} q$$

and

$$\sigma' \in [\![\, q \,]\!]_{\mathcal{S}} \ .$$

For partial correctness, we need to show that

$$\begin{aligned}[\![\, u := Svc(v) \,]\!]^{\pi}_{part,\mathcal{S}} \, (\, [\![\, \forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w] \,]\!]_{\mathcal{S}} \,) \\ \subseteq [\![\, q \,]\!]_{\mathcal{S}}\end{aligned}$$

holds. Otherwise, the proof is the same.

For total correctness, if $\sigma \in [\![\, p \,]\!]_{\mathcal{S}}$, then by (4.2), (4.3), and (4.4) there always exists σ' such that $\langle u := Svc(v), \sigma \rangle \to \langle E, \sigma' \rangle$.

For partial correctness, if $\sigma \in [\![\, \forall w \text{ s.t. } post_{Svc}[i := v, o := w] : q[u := w] \,]\!]_{\mathcal{S}}$ and $\sigma \notin [\![\, pre[i := v]_{\mathcal{S}} \,]\!]$, then $\langle u := Svc(v), \sigma \rangle \not\to \langle E, \sigma' \rangle$, and the workflow blocks: $[\![\, u := Svc(v) \,]\!]^{\pi}_{part,\mathcal{S}}(\sigma) = \emptyset$, and $\emptyset \subseteq [\![\, q \,]\!]_{\mathcal{S}}$.

Take

For total correctness, we need to show that

$$[\![\, take(a, A) \,]\!]^{\pi}_{tot,\mathcal{S}}([\![\, p \,]\!]_{\mathcal{S}}) \subseteq [\![\, q \,]\!]_{\mathcal{S}}$$

with $p := A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]$. Let $\sigma \in [\![\, p \,]\!]_{\mathcal{S}}$ and $\langle take(a, A), \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma' \rangle$. By semantics of *take*, we get for any $b \in A$:

$$\sigma' = \sigma[a := b, A := A \setminus \{b\}] \ .$$

From $\sigma \in [\![\, p \,]\!]_{\mathcal{S}}$, we get for all $b \in A$:

$$\sigma \models^{CR}_{K} q[a := b, A := A \setminus \{b\}],$$

therefore, by the Substitution Lemma (Apt et al., 2009, Lemma 2.4, p. 47),

$$\sigma[a := b, A := A \setminus \{b\}] \models^{CR}_{K} q$$

and

$$\sigma' \models_K^{CR} q \ .$$

For partial correctness, we need to show the same with $p := \forall b \in A : q[a := b, A := A \setminus \{b\}]$, and the proof is the same.

For total correctness, if $\sigma \in [\![\, p \,]\!]_{\mathcal{S}}$, then by the condition of the transition rule for *take* there always exists σ' such that $\langle \mathit{take}(a, A), \sigma\rangle \rightarrow \langle E, \sigma'\rangle$.

For partial correctness, if $\sigma \in [\![\, \forall b \in A : q[a := b, A := A \setminus \{b\}] \,]\!]_{\mathcal{S}}$ and $\sigma \notin [\![\, A \neq \emptyset \,]\!]_{\mathcal{S}}$, then $\langle \mathit{take}(a, A), \sigma\rangle \not\rightarrow \langle E, \sigma'\rangle$, and the workflow blocks: $[\![\, \mathit{take}(a, A) \,]\!]^{\pi}_{\mathit{part},\mathcal{S}}(\sigma) = \emptyset$, and $\emptyset \subseteq [\![\, q \,]\!]_{\mathcal{S}}$.

Foreach

To show: The premise of the `foreach` rule implies its consequence, that is:

$$\begin{array}{rl} & \models_K^{CR} \{p \wedge A \neq \emptyset\}\ \mathit{take}(a, A); W\ \{p\} \\ \text{implies} & \models_K^{CR} \{p\}\ \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}\ \{p \wedge A = \emptyset\} \end{array}$$

In contrast to `while` loops, `foreach` loops always have n executions of the loop body, with $|A| = n$, that is, the size of the set A, and thus always terminate. We include $|A| = n$ in the assertions. By the premise, the semantics of *take* and $A \notin \mathit{change}(W)$, we get

$$[\![\, \mathit{take}(a, A); W \,]\!]^{\pi}_{\mathcal{S}}([\![\, p \wedge A \neq \emptyset \wedge |A| = n \,]\!]_{\mathcal{S}}) \subseteq [\![\, p \wedge |A| = n - 1 \,]\!]_{\mathcal{S}} \ . \tag{4.8}$$

We use this property to show that $[\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, p \,]\!]_{\mathcal{S}}) \subseteq [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}}$. The proof proceeds by induction.

Base case $|A| = n = 0$: By semantics of `foreach`, we have

$$[\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}}) = [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}} \ . \tag{4.9}$$

Induction step $|A| = n + 1$, and thus $A \neq \emptyset$. We use monotonicity of semantics for this proof (Apt et al., 2009, Lemma 3.3, p. 62-63).

$$\begin{array}{l} \quad [\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}}) \\ = \{ \text{ by semantics of } \texttt{foreach} \text{ and } A \neq \emptyset \ \} \\ \quad [\![\, \mathit{take}(a, A); W; \texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}}) \\ = \{ \text{ by Def. of sequential composition } \} \\ \quad [\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, \mathit{take}(a, A); W \,]\!]_{\mathcal{S}}([\![\, p \wedge |A| = n + 1 \,]\!]_{\mathcal{S}})) \\ \subseteq \{ \text{ by (4.8) and monotonicity of semantics } \} \\ \quad [\![\texttt{foreach}\ a \in A\ \texttt{do}\ W\ \texttt{od}]\!]^{\pi}_{\mathcal{S}}([\![\, p \wedge \#A = n \,]\!]_{\mathcal{S}}) \\ \subseteq \{ \text{ by induction hypothesis } \} \\ \quad [\![\, p \wedge A = \emptyset \,]\!]_{\mathcal{S}} \end{array}$$

Rule of consequence By the premise of the rule, we have

$$[\![W]\!]^{\pi}_{\mathcal{S}}([\![r]\!]_{\mathcal{S}}) \subseteq [\![s]\!]_{\mathcal{S}} ,$$

as well as valid implications $p \Rightarrow r$ and $s \Rightarrow q$ in K, therefore

$$[\![W]\!]^{\pi}_{\mathcal{S}}([\![p]\!]_{\mathcal{S}}) \subseteq [\![W]\!]^{\pi}_{\mathcal{S}}([\![r]\!]_{\mathcal{S}}) \subseteq [\![s]\!]_{\mathcal{S}} \subseteq [\![q]\!]_{\mathcal{S}} .$$

□

Thus, a proof outline constructed according to the proof calculus indeed shows correctness of its workflow template for all possible service mappings π. The proof outline in Figure 4.3 shows that the FILTER template of Example 4.4 is indeed correct.

4.4.2.2 Proof of Completeness

As in the proof of Theorem 3.17 (Completeness of Proof Calculus), we prove completeness with an intermediate step by defining *weakest liberal preconditions* and *weakest preconditions*. Again, the important difference to the corresponding definitions in Chapter 3 is the use of the semantics of workflow *templates* including its parameterization with service mappings π.

Definition 4.17 (Weakest (Liberal) Preconditions)**.** Set WT be a workflow template and Φ be a set of states. We define the *weakest liberal precondition* of WT regarding Φ as follows:

$$wlp(WT, \Phi) = \{\sigma \mid [\![W]\!]^{\pi}_{part,\mathcal{S}}(\sigma) \subseteq \Phi\} .$$

Correspondingly, the *weakest precondition* of WT regarding Φ is:

$$wp(WT, \Phi) = \{\sigma \mid [\![W]\!]^{\pi}_{tot,\mathcal{S}}(\sigma) \subseteq \Phi\} .$$

Weakest (liberal) preconditions define sets of states. By the *Theorem of Definability* there exists a corresponding assertion for every precondition (Apt et al., 2009, Theorem 3.4, p. 87), and therefore we can represent weakest (liberal) preconditions with corresponding assertions (cf. page 52). As in Section 3.4.3, we follow the proof structure of Apt et al. and prove completeness in two steps:

(1) We show that $\vdash^{CR}_{K} \{wlp(WT, q)\}\ WT\ \{q\}$ holds for every wlp (and wp). We do this for the new service call, *take* and `foreach` statements.

(2) Apt et al. show that whenever a workflow is correct with respect to some p, q, then it is provably so.

```
$\{\forall z \in A : pre_{Acquire}(z)\}$
$\{A \subseteq A \land \big(\forall u \in A : pre_{Acquire}(u)\big) \land \emptyset = \{b \in A \setminus A \mid isTargetElement(b)\}\}$
$Z := A;$
$\{Z \subseteq A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land \emptyset = \{b \in A \setminus Z \mid isTargetElement(b)\}\}$
$B := \emptyset;$
$\{\mathbf{inv} : Z \subseteq A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land B = \{b \in A \setminus Z \mid isTargetElement(b)\}\}$
**foreach** $z \in Z$ **do**
    $\{Z \neq \emptyset \land \mathbf{inv}\}$
    $\{Z \neq \emptyset \land \forall x \in Z : \Big(pre_{Acquire}(x) \land x \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land (Z \setminus \{x\}) \subseteq$
    $A \land B \cup \{x\} = \{b \in A \setminus \big((Z \setminus \{x\}) \cup \{x\}\big) \mid isTargetElement(b)\} \cup \{x\}\Big)\}$
    $take(z, Z);$
    $\{pre_{Acquire}(z) \land z \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land Z \subseteq A \land B \cup \{z\} = \{b \in$
    $A \setminus (Z \cup \{z\}) \mid isTargetElement(b)\}\}$
    $y := Acquire(z);$
    $\{post_{Acquire}(z, y) \land z \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land Z \subseteq A \land B \cup \{z\} = \{b \in$
    $A \setminus (Z \cup \{z\}) \mid isTargetElement(b)\} \cup \{z\}$
    **if** $isTargetValue(y)$ **then**
        $\{isTargetElement(z) \land z \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land Z \subseteq A \land B \cup \{z\} =$
        $\{b \in A \setminus (Z \cup \{z\}) \mid isTargetElement(b)\} \cup \{z\}\}$
        $B := B \cup \{z\}$
        $\{isTargetElement(z) \land z \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land Z \subseteq A \land B = \{b \in$
        $A \setminus (Z \cup \{z\}) \mid isTargetElement(b)\} \cup \{z\}\}$
        $\{\mathbf{inv}\}$
    **else**
        $\{\neg isTargetElement(z) \land z \in A \land \big(\forall u \in Z : pre_{Acquire}(u)\big) \land Z \subseteq A \land B = \{b \in$
        $A \setminus (Z \cup \{z\}) \mid isTargetElement(b)\}\}$
        $\{\mathbf{inv}\}$
        $skip$
        $\{\mathbf{inv}\}$
    **fi**
    $\{\mathbf{inv}\}$
**od**
$\{Z = \emptyset \land \mathbf{inv}\}$
$\{B = \{b \in A \mid isTargetElement(b)\}\}$
```

FIGURE 4.3: Proof outline for correctness of the FILTER template

Again, as the proof of completeness (2) does not rely on workflow template statements directly, but on the general argument of $\vdash_K^{CR} \{wlp(WT, q)\}\ WT\ \{q\}$ (and wp), it is sufficient for this thesis to show that argument (1) is indeed true for the new workflow statements (cf. page 52 for a summary of the formal argument). As for workflows, we define and prove a Weakest (Liberal) Precondition Lemma to relate the weakest (liberal) preconditions of the new workflow template statements with assertions. Then, we prove $\vdash_K^{CR} \{wlp(WT, q)\}\ WT\ \{q\}$ (and wp) for the new statements. As this is step (1) of the overall proof of completeness, it concludes the proof of Theorem 4.16. For more details on the different steps of this proof and proofs for the standard workflow statements, we refer to Apt et al., 2009, p. 89-91.

Lemma 4.18 (Weakest (Liberal) Preconditions of Workflow Template Elements)**.** *We use variable, service, and assertion names as before, esp. with $S \in \mathcal{SVC}_K \cup \mathcal{SP}_K$ and $\pi : \mathcal{SVC}_K \to \mathcal{SP}_K$. Then the following statements hold:*

Service Call

$$\begin{array}{rl} & wlp(u := S(v), q) \\ \Leftrightarrow & \forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w] \end{array}$$

and

$$\begin{array}{rl} & wp(u := S(v), q) \\ \Leftrightarrow & \pi(pre_S[i := v]) \wedge \forall w : (\pi(post_S[i := v, o := w]) \Rightarrow q[u := w]) \end{array}$$

Take

$$\begin{array}{rl} & wlp(take(a, A), q) \\ \Leftrightarrow & \forall b \in A : q[a := b, A := A \setminus \{b\}] \end{array}$$

and

$$\begin{array}{rl} & wp(take(a, A), q) \\ \Leftrightarrow & A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \end{array}$$

Foreach

$$\begin{array}{rl} & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q) \wedge A \neq \emptyset \\ \Rightarrow & wlp(take(a, A); WT, wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q)) \end{array}$$

and

$$\begin{array}{rl} & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q) \wedge A = \emptyset \\ \Rightarrow & q \end{array}$$

and

$$\begin{array}{rl} & wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q) \\ \Leftrightarrow & wp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q) \end{array}$$

Proof. We prove the assertions for all three workflow statements separately. For *take* and loops, the proofs for *wlp* and *wp* are the same, and *wlp* can be substituted by *wp*. For the service call they are slightly different.

Service Call

For total correctness, by definition,

$$wp(u := S(v), [\![q]\!]_{\mathcal{S}}) = \{\sigma \mid [\![u := S(v)]\!]^{\pi}_{tot,\mathcal{S}}(\sigma) \subseteq [\![q]\!]_{\mathcal{S}}\}$$

and

$$\begin{aligned} & wp(u := S(v), q) \\ \Leftrightarrow\quad & \pi(pre_S[i := v]) \wedge \forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w] \ . \end{aligned}$$

Let $\sigma \in [\![\pi(pre_S[i := v]) \wedge \forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w]]\!]_{\mathcal{S}}$. Then, by the transition rule for service calls, there is $\langle u := S(v), \sigma\rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma'\rangle$ with $\sigma' \in [\![q]\!]_{\mathcal{S}}$.

If $\sigma \notin [\![\pi(pre_S[i := v]) \wedge \forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w]]\!]_{\mathcal{S}}$, then we distinguish two cases: If $\sigma \notin [\![\pi(pre[i := v]_S)]\!]_{\mathcal{S}}$, by the transition rule, the workflow blocks, that is, $[\![u := S(v)]\!]^{\pi}_{tot,\mathcal{S}}(\sigma) = \{\bot\}$, and $\{\bot\} \not\subseteq [\![q]\!]_{\mathcal{S}}$. If $\sigma \notin [\![\forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w]]\!]_{\mathcal{S}}$ (but $\sigma \in [\![\pi(pre[i := v]_S)]\!]_{\mathcal{S}}$), then there exist σ' and w such that $\sigma' \models^{CR}_{K} \pi(post_S[i := v, o := w]) \wedge \neg q[u := w]$, and $\langle u := S(v), \sigma\rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma'\rangle$, but $\sigma' \notin [\![q]\!]_{\mathcal{S}}$.

For partial correctness, by definition,

$$wlp(u := S(v), [\![q]\!]_{\mathcal{S}}) = \{\sigma \mid [\![u := S(v)]\!]^{\pi}_{part,\mathcal{S}}(\sigma) \subseteq [\![q]\!]_{\mathcal{S}}\}$$

and

$$\begin{aligned} & wlp(u := S(v), q) \\ \Leftrightarrow\quad & \forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w] \ . \end{aligned}$$

Let $\sigma \in [\![\forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w]]\!]_{\mathcal{S}}$. Then, we distinguish two cases: With $\sigma \in [\![\pi(pre[i := v]_S)]\!]_{\mathcal{S}}$, by the transition rule for service calls, there is $\langle u := S(v), \sigma\rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma'\rangle$ with $\sigma' \in [\![q]\!]_{\mathcal{S}}$. With $\sigma \notin [\![\pi(pre[i := v]_S)]\!]$, by the transition rule, the workflow blocks, that is, $[\![u := S(v)]\!]^{\pi}_{part,\mathcal{S}}(\sigma) = \emptyset$, and $\emptyset \subseteq [\![q]\!]_{\mathcal{S}}$.

If $\sigma \notin [\![\forall w : \pi(post_S[i := v, o := w]) \Rightarrow q[u := w]]\!]_{\mathcal{S}}$ (but $\sigma \in [\![\pi(pre[i := v]_S)]\!]_{\mathcal{S}}$), then there exist σ' and w such that $\sigma' \models^{CR}_{K} \pi(post_S[i := v, o := w]) \wedge \neg q[u := w]$, and $\langle u := S(v), \sigma\rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma'\rangle$, but $\sigma' \notin [\![q]\!]_{\mathcal{S}}$.

Take For total correctness, by definition,

$$wp(take(a, A), [\![q]\!]_{\mathcal{S}}) = \{\sigma \mid [\![take(a, A)]\!]^{\pi}_{tot,\mathcal{S}}(\sigma) \subseteq [\![q]\!]_{\mathcal{S}}\}$$

and

$$\begin{aligned} & wp(take(a, A), q) \\ \Leftrightarrow\quad & A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \ . \end{aligned}$$

Let $\sigma \in [\![\, A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}] \,]\!]_{\mathcal{S}}$ and σ' a state such that $\langle \mathit{take}(a, A), \sigma \rangle \rightarrow^{\pi}_{\mathcal{S}} \langle E, \sigma' \rangle$. From the transition axiom for *take* we know that whenever there is a b such that

$$\sigma'(b) \in \sigma(A)$$

then

$$\sigma'(A) = \sigma(A) \setminus \sigma'(b) \ ,$$

or, comparing the states directly,

$$\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \quad \sigma' = \sigma[A := A \setminus \{b\}] \ . \tag{4.10}$$

Because

$$\sigma \models^{CR}_{K} \big(A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]\big)$$

and therefore also

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \\ &\sigma \models^{CR}_{K} \big(A \neq \emptyset \wedge q[a := b, A := A \setminus \{b\}]\big)\end{aligned}$$

as well as

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \\ &\sigma \models^{CR}_{K} q[a := b, A := A \setminus \{b\}]\end{aligned}$$

by substitution

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \\ &\sigma[A := A \setminus \{b\}] \models^{CR}_{K} q[a := b] \ .\end{aligned}$$

Now by (4.10),

$$\begin{aligned}&\forall b \text{ s.t. } \sigma'(b) \in \sigma(A) : \\ &\sigma' \models^{CR}_{K} q[a := b] \ ,\end{aligned}$$

and because this holds for every b in the set Var, including the original a,

$$\sigma' \models^{CR}_{K} q$$

and therefore $\sigma' \in [\![\, q \,]\!]_{\mathcal{S}}$.

For partial correctness, by definition,

$$\begin{array}{rl} & \mathit{wlp}(\mathit{take}(a, A), q) \\ \Leftrightarrow & \forall b \in A : q[a := b, A := A \setminus \{b\}] \ . \end{array}$$

Let $\sigma \in [\![\forall b \in A : q[a := b, A := A \setminus \{b\}]]\!]_{\mathcal{S}}$. Then either $\sigma(A) \neq \emptyset$, and the proof works the same as for total correctness; or $\sigma = \emptyset$, and by the transition rule the workflow blocks, that is,

$$[\![\, \mathit{take}(a, A) \,]\!]^{\pi}_{\mathit{part},\mathcal{S}} \, (\sigma) = \emptyset \ ,$$

and $\emptyset \subseteq [\![\, q \,]\!]_{\mathcal{S}}$.

Foreach, empty set By definition of weakest (liberal) preconditions,

$$\mathit{wlp}(\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}, [\![\, q \,]\!]_{\mathcal{S}})$$

is

$$\{\sigma \mid [\![\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}]\!]^{\pi}_{\mathcal{S}} \, (\sigma) \subseteq [\![\, q \,]\!]_{\mathcal{S}}\}$$

and by semantics of `foreach` for an empty set $A = \emptyset$,

$$\subseteq \quad [\![\, q \,]\!]_{\mathcal{S}} \ .$$

Therefore

$$\mathit{wlp}(\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}, [\![\, q \,]\!]_{\mathcal{S}}) \cap [\![\, A = \emptyset \,]\!]_{\mathcal{S}} \subseteq [\![\, q \,]\!]_{\mathcal{S}} \ .$$

By the Theorem of Definability (Apt et al., 2009, Theorem 3.4, p. 87), we can write this as assertion:

$$\mathit{wlp}(\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}, q) \wedge A = \emptyset \Rightarrow q \ .$$

Foreach, non-empty set By definition of weakest (liberal) preconditions,

$$\mathit{wlp}(\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}, [\![\, q \,]\!]_{\mathcal{S}})$$

is

$$\{\sigma \mid [\![\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}]\!]^{\pi}_{\mathcal{S}} \, (\sigma) \subseteq [\![\, q \,]\!]_{\mathcal{S}}\}$$

and by semantics of `foreach` for a non-empty set $A \neq \emptyset$,

$$\begin{array}{l} \subseteq \quad \{\sigma \mid [\![\, \mathit{take}(a, A); \mathit{WT} \,]\!]^{\pi}_{\mathcal{S}} \, (\sigma) \subseteq [\![\, p \,]\!]_{\mathcal{S}}\} \cap [\![\, A \neq \emptyset \,]\!]_{\mathcal{S}} \\ \text{s.t. } [\![\, p \,]\!]_{\mathcal{S}} = \{\sigma' \mid [\![\texttt{foreach } a \in A \texttt{ do } \mathit{WT} \texttt{ od}]\!]^{\pi}_{\mathcal{S}} \, (\sigma') \subseteq [\![\, q \,]\!]_{\mathcal{S}}\} \ . \end{array}$$

Now, by induction hypothesis,

$$\subseteq \quad \{\sigma \mid [\![\, \mathit{take}(a, A);\, \mathit{WT} \,]\!]^{\pi}_{\mathcal{S}}\, (\sigma) \subseteq \mathit{wlp}(\texttt{foreach} \ldots, [\![\, q \,]\!]_{\mathcal{S}})\} \ ,$$

which is by definition of weakest (liberal) preconditions

$$\subseteq \quad \mathit{wlp}(\mathit{take}(a, A);\, \mathit{WT}, \mathit{wlp}(\texttt{foreach} \ldots, [\![\, q \,]\!]_{\mathcal{S}})) \ .$$

By the Theorem of Definability (Apt et al., 2009, Theorem 3.4, p. 87), we can write this as assertion:

$$\mathit{wlp}(\mathit{take}(a, A);\, \mathit{WT}, \mathit{wlp}(\texttt{foreach}\ a \in A\ \texttt{do}\ \mathit{WT}\ \texttt{od}, q)) \ .$$

□

Now, we show that $\vdash^{CR}_{K} \{\mathit{wlp}(\mathit{WT}, q)\}\ \mathit{WT}\ \{q\}$ (and $\vdash^{CR}_{K} \{\mathit{wp}(\mathit{WT}, q)\}\ \mathit{WT}\ \{q\}$) for all new preconditions. This concludes the proof of Theorem 4.16.

Again, from a general point of view, we need two different proof calculi to prove partial and total correctness of a workflow template (Apt et al., 2009, p. 70–71). However, as the proof rules are almost the same, we use only "the" proof calculus, with alternative rules for the `while` loop, depending on the notion of correctness we are aiming at, as we already do in the proof of Theorem 3.17.

Proof. We prove $\vdash^{CR}_{K} \{\mathit{wlp}(\mathit{WT}, q)\}\ \mathit{WT}\ \{q\}$ (and $\vdash_{K} \{\mathit{wp}(\mathit{WT}, q)\}\ \mathit{WT}\ \{q\}$) by induction, using workflows with proof axioms as base cases and workflows with proof rules as induction steps.

Service Call (wp) To show: $\vdash^{CR}_{K} \{\mathit{wp}(u := S(v), q)\}\, u := S(v)\, \{q\}$. This is by definition of the weakest precondition for service calls:

$$\vdash^{CR}_{K} \{\pi(\mathit{pre}_S[i := v]) \wedge \forall w : \pi(\mathit{post}_S[i := v, o := w]) \Rightarrow q[u := w]\}\, u := S(v)\, \{q\} \ ,$$

which is covered by the proof axiom for total correctness for service calls.

Service Call (wlp) To show: $\vdash^{CR}_{K} \{\mathit{wlp}(u := S(v), q)\}\, u := S(v)\, \{q\}$. This is by definition of the weakest liberal precondition for service calls:

$$\vdash^{CR}_{K} \{\forall w : \pi(\mathit{post}_S[i := v, o := w]) \Rightarrow q[u := w]\}\, u := S(v)\, \{q\} \ ,$$

which is covered by the proof axiom for partial correctness for service calls.

Take (wp) To show: $\vdash^{CR}_{K} \{\mathit{wp}(\mathit{take}(a, A), q)\}\, \mathit{take}(a, A)\, \{q\}$. By definition of weakest precondition for *take*, that is

$$\vdash^{CR}_{K} \{A \neq \emptyset \wedge \forall b \in A : q[a := b, A := A \setminus \{b\}]\}\, \mathit{take}(a, A)\, \{q\} \ ,$$

which is covered by the proof axiom for total correctness for *take*.

Take (wlp) To show: $\vdash_K^{CR} \{wlp(take(a,A),q)\}\ take(a,A)\ \{q\}$. By definition of weakest liberal precondition for *take*, that is

$$\vdash_K^{CR} \{\forall b \in A : q[a := b, A := A \setminus \{b\}]\}\ take(a,A)\ \{q\}\ ,$$

which is covered by the proof axiom for partial correctness for *take*.

Foreach To show:
$\vdash_K^{CR} \{wlp(\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}, q)\}\ \texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}\ \{q\}$. By induction hypothesis, we have

$$\vdash_K^{CR} \{wlp(take(a,A); WT, wlp(\texttt{for}\ldots, q))\}\ take(a,A);\ WT\ \{wlp(\texttt{for}\ldots, q)\}$$

$\Rightarrow$ by Def. 4.18 and rule of consequence

$$\vdash_K^{CR} \{wlp(\texttt{for}\ldots, q) \wedge A \neq \emptyset\}\ take(a,A);\ WT\ \{wlp(\texttt{for}\ldots, q)\}$$

$\Rightarrow$ by `foreach` proof rule

$$\vdash_K^{CR} \{wlp(\texttt{for}\ldots, q)\}\ \texttt{for}\ldots\ \{wlp(\texttt{for}\ldots, q) \wedge A = \emptyset\}$$

$\Rightarrow$ by Def. 4.18, rule of consequence, and $A = \emptyset$

$$\vdash_K^{CR} \{wlp(\texttt{for}\ldots, q)\}\ \texttt{for}\ldots\ \{q\}$$

□

As it was already the case for workflows, in our context it is not necessary to prove termination of loops additionally, as `foreach` loops terminate by definition, because their semantics include the reduction of the (finite) iteration set.

4.5 Correct Instantiations

Up to now, we defined correctness both for a service composition and a composition template. Correctness of a template is defined over all possible service mappings, but a template is not a service composition itself, even if we replace the service placeholders using a service mapping: A template has its own workflow language with its own semantics, and it has additional constraint rules. To create service compositions with templates as blueprint (which is their original motivation), we need to formalize an *instantiation*, which maps workflow templates to workflows. This instantiation can happen within one domain, but also map a template from an abstract domain to a concrete target domain. As constraint rules are part of templates, but not of service compositions, they have to be treated differently. They serve two main purposes:

(1) They capture the influence of a service placeholder – or, more precisely, the effect of the service that should replace that placeholder.

(2) They are used as a set of *facts* in the definition of correctness of a template.

For service compositions, the only collection of facts is the domain knowledge. Therefore, constraint rules of a template have to be translated into terms built upon the vocabulary of the target knowledge base. This happens implicitly, as service placeholders are replaced with services, and their pre- and postcondition predicates with terms from the service description. Thus, constraint rules also become concrete.

The constraint rules are essential to the correctness definition of a template. A key question is how to move this importance to an instantiation (which has no constraints) and the target domain (which is fixed and cannot be modified). The main idea is that it is sufficient to ensure that constraint rules are *implied by* the existing domain knowledge, once instantiated. With this in mind, we will see that it is often not necessary to have an exhaustive domain knowledge to prove correctness of a template. Instead, the domain knowledge of a template may contain just the knowledge which is necessary to prove the template's correctness. Especially if the task at hand is abstract (compare the generic filter service, → Example 4.4), the necessary domain knowledge is small. We can even *create* a specialized knowledge base just to prove correctness of the template, which is completely independent of any other domain. This is what we did in the abstract filter example, and this section draws the missing link between the domain specific examples and the abstract template example.

Before defining an instantiation, we define an *ontology mapping*. This way, it is possible to map a small, abstract, template-related ontology to a large, domain-specific one. Using this ontology mapping, we define an instantiation. As a result, we come up with the following side condition: During instantiation, the constraint rules of a template have to comply with the domain knowledge, that is, the "target" of the ontology mapping. If this side condition is respected, the resulting service composition is correct by construction. We formalize this as a theorem and give a proof arguing over the interpretations of the logical structures.

4.5.1 Ontology Mappings

A knowledge base in terms of an ontology describes the *vocabulary* of a domain, that is, domain specific concepts, as well as their relations. Mapping one ontology to another contains therefore a mapping of every concept to its counterpart in the target ontology. The same is true for roles (predicates). Ontology mapping is not trivial, but a topic of research on its own. *Ontology conflicts* occur typically in complex ontologies, where concepts cannot be mapped to direct counterparts (Kumar and Harding, 2013, Noy, 2009). We assume that ontology conflicts do not occur and a mapping is possible, whether automatically or manually. We think this assumption is reasonable for both of our “abstract task pattern” scenarios: If a template is designed by domain experts for recurring tasks within a domain, the mapping is not necessary (that is, it is trivial); if the task is abstract and the template ontology is generated just for the purpose of template modeling, it will be typically quite small compared to a domain ontology, resulting in a low probability for ontology conflicts.

Definition 4.19 (Ontology Mapping)**.** Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ a domain ontology. Then $K_T \rhd_f K_D$ is a *signature homomorphous ontology mapping* from K_T to K_D by f, if f is a pair of mappings $f = (f_C : C_T \to C_D, f_P : P_T \to P_D)$ such that

- f_P preserves signatures with respect to f_C, that is $\forall p \in P_T$ with $p : T_1 \times \cdots \times T_n$ we have $f_P(p) : f_C(T_1) \times \cdots \times f_C(T_n)$;
- f preserves the rules R_T, that is $\forall r \in R_T$ there is $f(r) \in R_D$ with

$$f(r) = \begin{cases} f_P(P)(x,y) & \text{if } r = P(x,y) \text{ and } P \in P_T \\ f(b_1) \vee f(b_2) & \text{if } r = b_1 \vee b_2 \\ \neg f(b) & \text{if } r = \neg b\ . \end{cases}$$

We use f to denote the use of the “appropriate” f_C or f_P.

The following example maps the generic FILTER ontology ($\to$ Example 4.3) to the Tourism ontology excerpt ($\to$ Example 2.1).

Example 4.5 (Mapping Filter to Tourism Ontology)**.** *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be the template ontology from Example 4.3, and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ the domain ontology from Example 2.1. We define a mapping $f = (f_C, f_P)$ from K_T to K_D with $K_T \rhd_f K_D$ as:*

$$\begin{aligned} f_C :\ & \textsc{Element} \mapsto \textsc{Restaurant}\ , \\ & \textsc{Value} \mapsto \textsc{Rating}\ , \\ f_P :\ & \textsc{hasValue} \mapsto \textsc{hasRating}\ , \\ & \textsc{isTargetValue} \mapsto \textsc{isMinRating}\ , \\ & \textsc{isTargetElement} \mapsto \textsc{goodRestaurant}\ . \end{aligned}$$

Since the template ontology has no rules besides the functionality of predicates, f trivially preserves them. For our restaurant ontology, we assume the service GetRating to provide a lookup service for ratings of restaurants.

4.5.2 Template Instantiation

Mapping one ontology to another replaces the vocabulary used in the template with vocabulary from the domain. By applying this mapping to a service description $S^{Desc} =$

$(S^{Sig}, I_S, O_S, pre_S, post_S)$, we already get an "instantiated" description. This is especially true for templates, as their description relies on the abstract vocabulary of a (probably artificial) template ontology. As a next step, we replace the service placeholders. In our definition of composition templates we already use a service mapping ($\rightarrow$ Definition 4.3), but it maps placeholders to services within the same domain. Now, we define a service mapping, or *concretion*, to replace placeholders with services from the target domain. Consequently, this mapping makes use of the ontology mapping defined above.

Definition 4.20 (Service Placeholder Concretion)**.** Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ be ontologies with $K_T \rhd_f K_D$. Let $\mathcal{SP}_T$ be the set of service placeholders formalized using K_T, and $\mathcal{SVC}_D$ be the set of service descriptions formalized using K_D. Then $\pi_f : \mathcal{SP}_T \rightarrow \mathcal{SVC}_D$ is a *concretion of service placeholders from K_T to K_D*, if it respects signatures with respect to f, that is,

$$\begin{aligned} \text{if } \pi_f(sp) = svc \text{ and } sp : T_1 \rightarrow T_2 \ , & \\ & \text{then } svc : T_1' \rightarrow T_2' \\ & \text{with } f_C(T_1) \sqsubseteq_D T_1' \text{ and } T_2' \sqsubseteq_D f_C(T_2) \ . \end{aligned}$$

In this definition, π_f maps service placeholders to services. Additionally, we want to replace pre- and postcondition *predicates*, coming from service placeholders, with pre- and postcondition *formulas*, coming from services. Therefore, we lift the definition of π_f to formulas in general.

Definition 4.21 (Placeholder Predicate Replacement with Ontology Mapping)**.** Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ be ontologies with $K_T \rhd_f K_D$. Let $\mathcal{SP}_T$ be the set of service placeholders formalized using K_T, and $\mathcal{SVC}_D$ be the set of service descriptions formalized using K_D, and let $\pi_f : \mathcal{SP}_T \rightarrow \mathcal{SVC}_D$ be a service placeholder concretion. We lift π_f to $\Phi_T^{sp} \rightarrow \Phi_D$ such that for $\varphi \in \Phi_T^{sp}$

$$\pi_f(\varphi) = \begin{cases} pre_{\pi_f(sp)} & \text{if } \varphi = pre_{sp} \\ post_{\pi_f(sp)} & \text{if } \varphi = post_{sp} \\ f_P(P)(x,y) & \text{if } \varphi = P(x,y) \text{ and } P \in \mathcal{P}_T \\ F(x,y) & \text{if } \varphi = F(x,y) \text{ and } F \in \mathcal{F}_T \\ \neg \pi_f(\varphi_1) & \text{if } \varphi = \neg \varphi_1 \\ \pi_f(\varphi_1) \vee \pi_f(\varphi_2) & \text{if } \varphi = \varphi_1 \vee \varphi_2 \ . \end{cases}$$

The service placeholder concretion does not require an *exact* mapping of input and output types. Instead, it makes use of the subtype relation $\sqsubseteq_D$ of the target ontology K_D. The "direction" of subtyping follows closely the Liskov substitution principle, and, consequently, the ideas of covariance and contravariance from object-oriented programming (Liskov and Wing, 1994). The Liskov principle requires, in object-oriented programming, that everywhere where a general class (or method) is used, a more specific

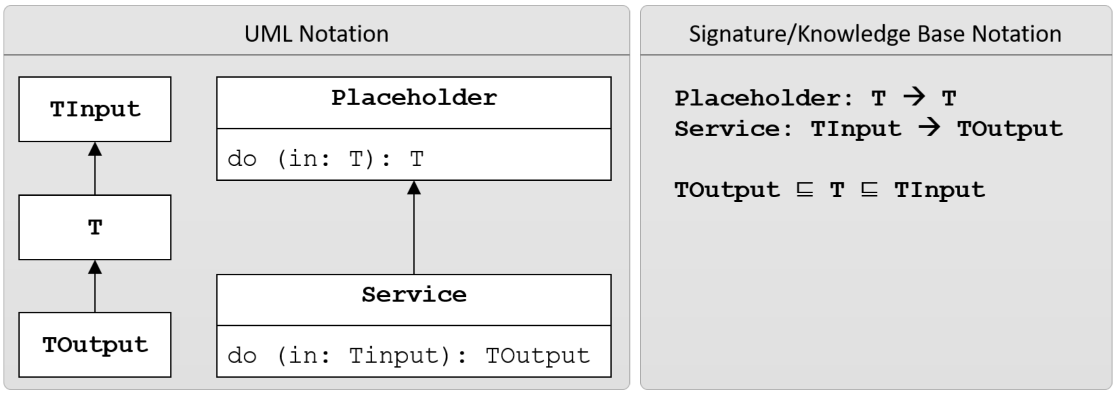

Figure 4.4: Liskov substitution principle with co- and contravariance for classical programming language specification and with service placeholders and services

class (or method) must be usable as well. Applied to our context, a service placeholder *sp* corresponds to a general class, and a concrete service *svc* to a specific class.

Whether or not the substitution principle is respected depends on the subtyping direction of the input and output parameters. In object-oriented programming, these are the parameters and return type(s) of a method; here, these are the inputs and outputs of services and placeholders. To respect the substitution principle, the input parameter types of *svc* have to be *contravariant* to their *sp* counterparts, that is, they may be more general, but not more specific. Similarly, the output parameter types have to be *covariant* to the *sp*'s outputs, that is, they may be more specific, but not more general. Figure 4.4 visualizes this relationship in comparison to object-oriented programming: On the left, a type hierarchy and a subclass relation is shown, with *Placeholder* as the super- and *Service* as the subclass. The superclass uses type T both as input and output type, while the subclass uses type *TInput* (the more general type) as input and type *TOutput* (the more specific type) as output, thus following the Liskov principle. On the right hand side, the equivalent situation is shown using the notation of this thesis. Example 4.6 illustrates this principle using our running example.

Example 4.6 (Direction of Subtyping in Service Concretions)**.** *Assume the following service signatures (without conditions) in the Tourism domain as given in Example 2.1.*

Placeholder
$\mathit{Acquire} : \mathit{Element} \rightarrow \mathit{Value}$
Service
$\mathit{GetRating} : \mathit{Restaurant} \rightarrow \mathit{Rating}$
$\mathit{GetStars} : \mathit{Restaurant} \rightarrow \mathit{Michelin}$
$\mathit{GetInfo} : \mathit{SnackBar} \rightarrow \mathit{InfoTag}$

The service GetRating *is, according to the definition of service mapping, a valid substitution for the service placeholder* Acquire*. The same is true for* GetStars*, as its output* *type,* Michelin*, is a subtype of the (required)* Rating*. In contrast,* GetInfo *is no valid substitution. Its input,* SnackBar *is a subtype of* Restaurant*, though for inputs only more general types are allowed. Also, its output,* InfoTag*, is more general than* Rating*, but for outputs only more specific types are allowed.*

Up to now, we defined service templates, ontology mappings, and service placeholder concretions. If these three are fixed, we can create a new service composition. Instantiating a template consists of two major steps: (1) Creating a new description based on the ontology mapping, and (2) creating a workflow from the workflow template by applying the service placeholder concretion π_f. As a result, we get a new service composition. This composition is not necessarily correct; correctness depends on the constraints rules. The next sections gives the details.

Definition 4.22 (Template Instantiation)**.** Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be the template's domain ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ the target domain ontology, with $K_T \rhd_f K_D$. Let (Sct^{Desc}, WT, CR) be a composition template with $Sct^{Desc} = (Sct^{Sig}, I_{Sct}, O_{Sct}, pre_{Sct}, post_{Sct})$ and $Sct^{Sig} = Sct : T_1 \times \cdots \times T_k \rightarrow T_{k+1} \times \cdots \times T_n$. Let $\pi_f : \mathcal{SP}_T \rightarrow \mathcal{SVC}_D$ be a service placeholder concretion.

The *template instantiation* of Sct using π_f results in a service composition Sc, which is defined as

$$(Sc^{Desc}, W)$$

with

$$Sc^{Desc} = \quad (Sc^{Sig}, I_{Sc}, O_{Sc}, pre_{Sc}, post_{Sc})$$

such that

$$Sc^{Sig} = \quad Sc : f_C(T_1) \times \cdots \times f_C(T_k) \rightarrow f_C(T_{k+1}) \times \cdots \times f_C(T_n)$$

and

$$\begin{aligned}
Sc &= \text{a new unique name} \\
I_{Sc} &= \big\{x \mid x' \in I_{Sct} \wedge name(x) = name(x') \wedge type(x) = f_C(x')\big\} \\
O_{Sc} &= \big\{x \mid x' \in O_{Sct} \wedge name(x) = name(x') \wedge type(x) = f_C(x')\big\} \\
pre_{Sc} &= \pi_f(pre_{Sct}) \\
post_{Sc} &= \pi_f(post_{Sct})
\end{aligned}$$

where Sc is a new name for the composition. The resulting workflow W is defined as

$$\begin{aligned}
\pi_f(\mathit{skip}) :=\ \mathit{skip} \qquad & \pi_f(u := t) :=\ u := t \\
\pi_f(u := \mathit{Svc}(i)) :=\ & u := \pi_f(\mathit{Svc})(i) \\
\pi_f(W_1; W_2) :=\ & \pi_f(W_1); \pi_f(W_2) \\
\pi_f(\textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}) :=\ & \textbf{if } \pi_f(B) \textbf{ then } \pi_f(W_1) \textbf{ else } \pi_f(W_2) \textbf{ fi} \\
\pi_f(\textbf{while } B \textbf{ do } W \textbf{ od}) :=\ & \textbf{while } \pi_f(B) \textbf{ do } \pi_f(W) \textbf{ od} \\
\pi_f(\textbf{foreach } a \in A \textbf{ do } W \textbf{ od}) :=\ & \textbf{foreach } a \in A \textbf{ do } \pi_f(W) \textbf{ od}\ .
\end{aligned}$$

Formally, workflows and workflow templates have different semantics, though they are equivalent.

Lemma 4.23 (Same Semantics of Workflow Templates and Instantiations)**.** *Let WT be a workflow template and $\pi : \mathcal{SP}_K \to \mathcal{SVC}_K$ an instantiation with services over ontology K. Then the following holds:*

$$\forall \textit{ logical structures } \mathcal{S} : \quad [\![\, WT \,]\!]^{\pi}_{\mathcal{S}} = [\![\, \pi(WT) \,]\!]_{\mathcal{S}}\ .$$

Proof. The semantics of workflows and workflow templates are both defined on transitions (→ Definitions 3.8 and 4.7). The only difference is the application of instantiation parameter π. The instantiation replaces service placeholders (and, consequently, their pre- and postcondition predicates) with services (and concrete pre- and postcondition formulas). Thus, the Lemma follows directly from Definitions 3.8, 4.7 and 4.22. □

The following example takes the abstract FILTER template of Example 4.4 and instantiates it into the Tourism domain of Example 2.1.

Example 4.7 (Restaurant Quality Filter Instantiation)**.** *The* FILTER *template accepts a set of input and delivers the subset that matches a filter criterion. We use the following instantiation to create a service composition which filters set of restaurant data by a quality predicate. We use the ontology mapping from Example 4.5 and the service mapping $\pi_f(Acquire) = GetRating$. Figure 4.5 shows the details.*

The next example creates a similar instantiation. This time, the filter criterion is still the rating of a restaurant, but the instantiated service retrieves the price, which leads to a service composition which fails to fulfill the correctness definition given in Definition 3.12.

Example 4.8 (Restaurant Price Filter Instantiation)**.** *Again, we use the ontology mapping from Example 4.5, now with the service mapping $\pi_f(Acquire) = GetPrice$. Figure 4.6 shows the details.*

Now, the price acquired by the service *GetPrice* is used as parameter for the predicate *isMinRating*. As we can see from these examples, a syntactically correct service composition, created from a correct template, is not automatically correct itself. However,

Composition	FilterRestaurantByRating
Inputs	A : **set** Restaurant
Outputs	B : **set** Restaurant
Precond.	$\forall a \in A : pre_{GetRating}(a)$
Postcond.	$B = \{b \in A \mid goodRestaurant(b)\}$

Workflow

```
$Z := A$;
$B := \emptyset$ ;
**foreach** $z \in Z$ **do**
    $y := GetRating(z)$ ;
    **if** $isMinRating(y)$ **then**
        $B := B \cup \{z\}$
    **else**
        *skip*
    **fi**
**od**
```

Figure 4.5: Service composition to filter a set of restaurants, using a filter *goodRestaurant* and a service *GetRating*

if we respect certain side conditions – the instantiated constraint rules in relation to the target ontology – we can indeed create a composition, which is provably correct by construction. The next sections elaborates on the details.

4.5.3 Correctness by Construction

As we have seen, deriving a service composition from a provably correct template does not yield a correct result automatically. The main issue is the treatment of the constraint rules of the template. The correctness definition of templates ($\rightarrow$ Definition 4.11) uses the constraints as facts, that is, a template can be proven correct *under the assumption that the constraints hold.*

The correctness definition of service compositions ($\rightarrow$ Definition 3.12) does not contain any corresponding concept, which is why constraints are not part of the template instantiation directly ($\rightarrow$ Definition 4.22). The constraints CR are, however, translated from the abstract domain K_T to the target domain $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$, resulting in $\pi_f(CR)$. Now, there are two possible scenarios. At first, the resulting constraints are fully compatible with the knowledge of the target domain and the constraints always hold in the target ontology, that is, $\forall \mathcal{S} : \mathcal{S} \models_D \pi_f(CR)$. At second, for some logical structures the constraints contradict the existing domain knowledge, that is, $\exists \mathcal{S} : \big(\mathcal{S} \models_D R_D\big) \wedge \big(\mathcal{S} \not\models_D \pi_f(CR)\big)$.

We defined correctness over all logical structures which comply to the domain knowledge in question. We are therefore interested in the first case, and state a theorem relating

Composition	FilterRestaurantByRatingByPrice
Inputs	A : **set** Restaurant
Outputs	B : **set** Restaurant
Precond.	$\forall a \in A : pre_{GetPrice}(a)$
Postcond.	$B = \{b \in A \mid goodRestaurant(b)\}$

Workflow

```
$Z := A$;
$B := \emptyset$ ;
**foreach** $z \in Z$ **do**
    $y := GetPrice(z)$ ;
    **if** $isMinRating(y)$ **then**
        $B := B \cup \{z\}$
    **else**
        *skip*
    **fi**
**od**
```

Figure 4.6: Service composition to filter a set of restaurants, using a filter *goodRestaurant* and a service *GetPrice*

the validity of the constraint rules and the correctness of the generated composition: Whenever the (instantiated) constraint rules comply with the target domain (that is, they are valid for every logical structure which complies with the domain), then the composition is also correct.

Theorem 4.24 (Constraint Rule Compliance)**.** *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ be a domain ontology with $K_T \rhd_f K_D$. Let (Sct^{Desc}, WT, CR) be a* correct *service composition template, that is, $\models_T \{pre_{Sct}\}\, Sct\, \{post_{Sct}\}$. Let $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ be a concretion of service placeholders and (Sc^{Desc}, W) the service composition created by π_f. If the concretized constraint rules of the template comply with the domain ontology, then the service composition obtained from $\pi_f(Sct)$ is correct, that is,*

$$\begin{array}{rl} \textit{if} & \models_T \{pre_{Sct}\}\, Sct\, \{post_{Sct}\} \\ \textit{and} & R_D \models \pi_f(CR) \\ \textit{then} & \models_D \{pre_{Sc}\}\, Sc\, \{post_{Sc}\} \ . \end{array}$$

The proof of this theorem relies on the interpretations $\mathcal{I}$ of predicates of the ontologies and how they are constrained by the concretized constraint rules. To give a formal proof, we therefore need to state some additional relations between logical structures and the various mappings.

Up to now, we established a *syntactical* correspondence between ontologies. However, the workflow semantics rely on logical structures, and the correctness of workflows is defined

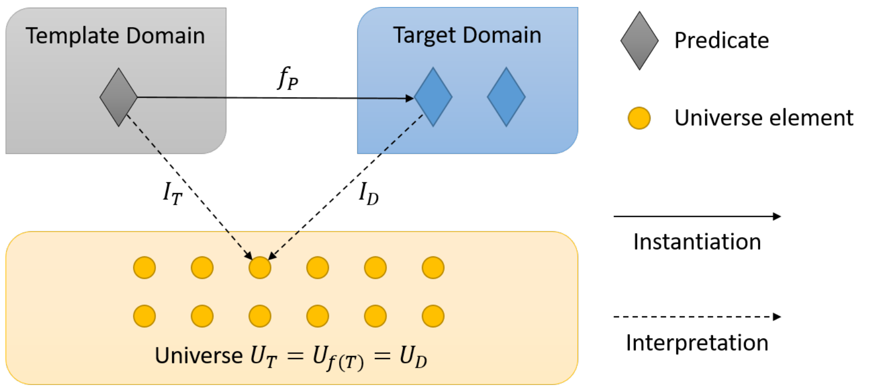

FIGURE 4.7: By logical structure correspondence, the interpretations of predicates of the abstract domain and their mapped counterparts of the target domain are the same

over logical structures as well. Therefore, we firstly establish a *semantical* relation between two ontologies $K_T \rhd_f K_D$. Starting from a given logical structure for which the rules of the target ontology hold, we can always construct a corresponding structure in the source ontology. The following proposition formalizes this claim.

Proposition 4.25 (Logical Structure Correspondence). *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ be ontologies with $K_T \rhd_f K_D$, and let $\mathcal{S}_D = (\mathcal{U}_D, \mathcal{I}_D)$ be a logical structure over K_D. If $\mathcal{S}_D \models R_D$, then we can construct a corresponding logical structure $\mathcal{S}^{\rhd f}$, where*

$$\mathcal{S}^{\rhd f} = (\mathcal{U}_T, \mathcal{I}_T), \textit{ with } \begin{array}{l} \mathcal{U}_T = \mathcal{U}_{f(T)} \textit{ and} \\ \mathcal{I}_T(p) = \mathcal{I}_D(f_P(p)) \textit{ for } p \in \mathcal{P}_T \end{array}$$

such that $\mathcal{S}^{\rhd f} \models R_T$.

Proof. We construct $\mathcal{S}^{\rhd f}$ based on $\mathcal{S}_D$, using the identical universe and corresponding interpretations up to f. By Definition 4.19 (Ontology Mapping), f is rule preserving. As $\mathcal{I}_T$ and $\mathcal{I}_D$ have identical results, for all rules $r \in R_T$ we have $\mathcal{S}^{\rhd f} \models r$ if and only if $\mathcal{S}_D \models f(r)$. Because all source ontology rules are translated to the target ontology, that is, $f(R_T) \subseteq R_D$, the proposition holds. Figure 4.7 visualizes the core idea of this proof. $\square$

Constraint rules do not only contain predicates from the ontology, but also predicates representing the pre- and postconditions of service placeholders used in the template. Proposition 4.25 relates only logical structures which give interpretations for ontology predicates. The definitions of correctness for templates ($\rightarrow$ Definition 4.11) uses a mapping $\pi : \mathcal{SP}_T \rightarrow \mathcal{SVC}_T$, which maps service placeholders to the range of possible service descriptions using K_T. To prove Theorem 4.24, we also need a relation between π (used in the correctness of templates) and π_f (based on $\rhd_f$ and used in template instantiations). Again, the general idea is to construct a mapping π for a given concretion π_f.

Proposition 4.26 (Placeholder Mapping Correspondence). *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ a target domain ontology with $K_T \rhd_f K_D$. Let $\mathcal{S}_D = (\mathcal{U}_D, \mathcal{I}_D)$ be a logical structure over K_D. Let $\Psi \in \Phi_T^{sp}$ be a formula containing placeholders from $\mathcal{SP}_T$, and let $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ be a concretion with (we use superscripts T and D to refer to types derived von K_T and K_D)*

$$\begin{aligned}
&sp \in \mathcal{SP}_T \textit{ with}\\
&sp : T_1^T \times \cdots \times T_k^T \to T_{k+1}^T \times \cdots \times T_n^T \ ,\\
&svc \in \mathcal{SVC}_D \textit{ with}\\
&svc : T_1^D \times \cdots \times T_k^D \to T_{k+1}^D \times \cdots \times T_n^D \ ,\\
&\forall i \textit{ with } 0 < i \leq k : f_C(T_i^T) \sqsubseteq T_i^D \ ,\\
&\forall i \textit{ with } k < i \leq n : T_i^D \sqsubseteq f_C(T_i^T) \ ,\\
&\pi_f(sp) = svc \ .
\end{aligned}$$

In other words, svc follows the Liskov substitution principle regarding sp, that is, its input types are contravariant to the input types of sp, and its output types are covariant to the output types of sp. If $\mathcal{S}_D$ satisfies the rules of K_D and the concretized formula $\pi_f(\Psi)$, that is,

$$\mathcal{S} \models R_D \wedge \pi_f(\Psi) \ ,$$

then we can construct a corresponding concretion $\pi_f^{\rhd f} : \mathcal{SP}_T \to \mathcal{SVC}_T$ within the template ontology in the following way such that

$$\begin{aligned}
&svc' \in \mathcal{SVC}_T \ ,\\
&svc : T_1^T \times \cdots \times T_k^T \to T_{k+1}^T \times \cdots \times T_n^T \ ,\\
&\pi_f^{\rhd f}(sp) = svc' \ ,
\end{aligned}$$

where svc and svc′ refer to the same name; then we also know, that

$$\begin{aligned}
pre_{svc'} &\in \Phi_T \textit{ such that } \pi_f(pre_{svc'}) = pre_{svc} \ ,\\
post_{svc'} &\in \Phi_T \textit{ such that } \pi_f(post_{svc'}) = post_{svc} \ ;
\end{aligned}$$

and we can conclude $\mathcal{S}^{\rhd f} \models \pi_f^{\rhd f}(\Psi)$.

Proof. Consider some state $\sigma \models_{\mathcal{S}} \pi_f(\Psi)$ with $\Psi \in \Phi_T^{sp}$. Then, the interpretations $\mathcal{I}_D$ are defined for every predicate. We can construct $\mathcal{S}^{\rhd f}$ by Proposition 4.25 (Logical Structure Correspondence), where the interpretations of template predicates are by construction the same as the interpretations of the corresponding (by f) domain predicates. The only predicates without interpretations are the pre- and postconditions of placeholders. We can construct $\pi_f^{\rhd f}$ such that $\pi_f(\pi_f^{\rhd f}(pre_{sp})) = \pi_f(pre_{sp})$ (same for postcondition). By definition, the interpretations are then mapped to the corresponding predicates,

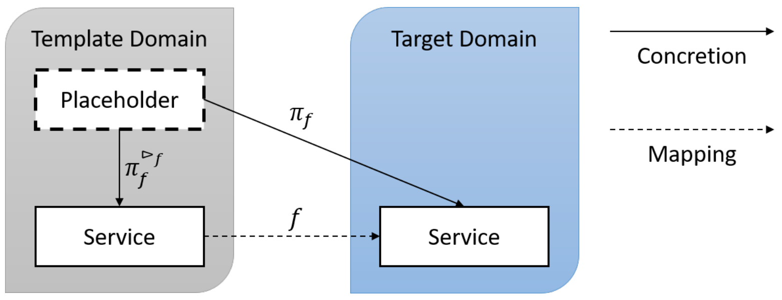

FIGURE 4.8: By placeholder mapping correspondence, intra-domain and inter-domain placeholder concretions correspond by using the same mapping

and $\sigma \models_{\mathcal{S}} \pi_f(\Psi) \Rightarrow \sigma \models_{\mathcal{S}^{\triangleright_f}} \pi_f^{\triangleright_f}(\Psi)$. The same is true for $\sigma \not\models_{\mathcal{S}} \pi_f(\Psi)$, therefore $\sigma \models_{\mathcal{S}} \pi_f(\Psi) \Leftrightarrow \sigma \models_{\mathcal{S}^{\triangleright_f}} \pi_f^{\triangleright_f}(\Psi)$. Figure 4.8 visualizes the core idea of this proof. □

Proposition 4.25 (Logical Structure Correspondence) relates logical structures, and Proposition 4.26 (Placeholder Mapping Correspondence) relates placeholder concretions (with and without an ontology mapping f of $\triangleright_f$). Given a formula $\Psi \in \Phi_T^{sp}$, that is, a formula based on the template ontology K_T and service placeholder predicates, we can conclude that we arrive at the same set of states, whether we use the service placeholder concretion π_f or the constructed $\pi_f^{\triangleright_f}$ to instantiate Ψ.

Lemma 4.27 (Same Set of States)**.** *Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ a domain ontology with $K_T \triangleright_f K_D$. Let $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ be a placeholder concretion, let $\Psi \in \Phi_T^{sp}$ be a formula containing pre- and postcondition placeholders from $\mathcal{SP}_T$, and let $\mathcal{S} \models \pi_f(\Psi)$; then*

$$[\![\pi_f(\Psi)]\!]_{\mathcal{S}} = [\![\pi_f^{\triangleright_f}(\Psi)]\!]_{\mathcal{S}^{\triangleright_f}} .$$

Proof. From the proof of Proposition 4.26 (Placeholder Mapping Correspondence) we already know that for any state σ the following holds:

$$\big(\sigma \models_{\mathcal{S}} \pi_f(\Psi)\big) \Leftrightarrow \big(\sigma \models_{\mathcal{S}^{\triangleright_f}} \pi_f^{\triangleright_f}(\Psi)\big)$$

By Definition 2.11 (Sets of States), the Lemma holds. □

The core statement of Theorem 4.24 (Constraint Rule Compliance) is that whenever an instantiation π_f with an ontology mapping $K_T \triangleright_f K_D$ is in accordance with the rules of the target ontology, especially the instantiation of the constraint rules of the template, then the resulting service composition is automatically correct. With the help of these propositions and lemmata given up to now, we are able to proof Theorem 4.24. Figure 4.9 gives an overview of the proof: First, we select a structure which satisfies the concretized constraints of the template (1). Then, we construct the corresponding structure which satisfies the original constraints (2). By definition, we know the semantics of the template (3) and can conclude that its instantiation has the same semantics (4). As the template is correct, the instantiation is also correct (5). We now give the formal proof.

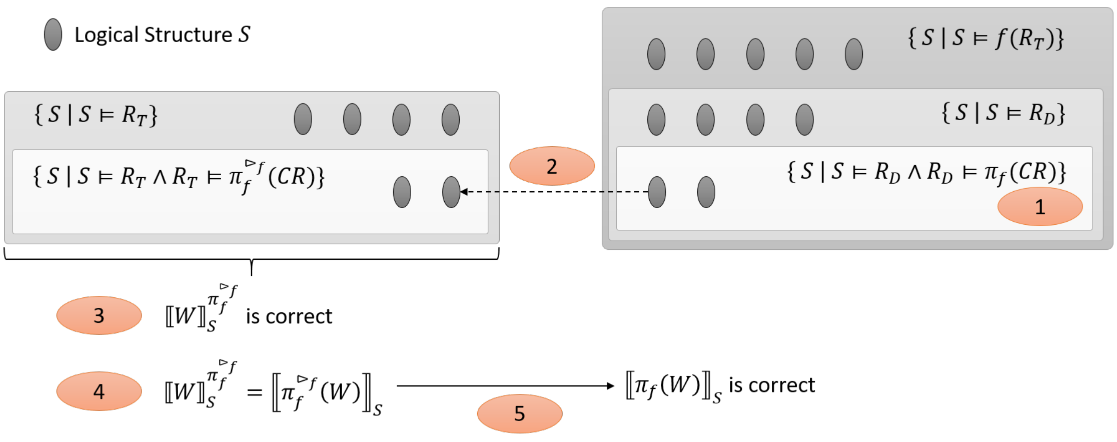
Logical Structure S
$\{S \mid S \vDash R_T\}$
$\{S \mid S \vDash R_T \wedge R_T \vDash \pi_f^{\rhd f}(CR)\}$
$\{S \mid S \vDash f(R_T)\}$
$\{S \mid S \vDash R_D\}$
$\{S \mid S \vDash R_D \wedge R_D \vDash \pi_f(CR)\}$
1
2
3
$[\![W]\!]_S^{\pi_f^{\rhd f}}$ is correct
4
$[\![W]\!]_S^{\pi_f^{\rhd f}} = [\![\pi_f^{\rhd f}(W)]\!]_S$
5
$[\![\pi_f(W)]\!]_S$ is correct

FIGURE 4.9: Graphical overview of the five steps to prove Theorem 4.24

Proof. Let $K_T = (C_T, P_T, \sqsubseteq_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, \sqsubseteq_D, R_D)$ a domain ontology with $K_T \rhd_f K_D$. Let (Sct^{Desc}, WT, CR) be a correct service composition template over K_T, and let $\pi_f : \mathcal{SP}_T \rightarrow \mathcal{SVC}_D$ be a concretion. We have to show that for an arbitrary concretion π_f, as long as the target ontology satisfies the concretized constraint rules of the template, the resulting composition $\pi_f(Sct) = (Sc^{Desc}, W)$ is indeed correct. Formally, for all structures $\mathcal{S}_D$ and concretions π_f the following has to hold:

$$\mathcal{S}_D \models R_D \wedge R_D \models \pi_f(CR) : [\![\pi_f(WT)]\!]_{\mathcal{S}_D}([\![\pi_f(pre_{Sct})]\!]_{\mathcal{S}_D}) \subseteq [\![\pi_f(post_{Sct})]\!]_{\mathcal{S}_D} ,$$

or, using (Sc^{Desc}, W) instead of $\pi_f(Sct)$:

$$\mathcal{S}_D \models R_D \wedge R_D \models \pi_f(CR) : [\![W]\!]_{\mathcal{S}_D}([\![pre_{Sc}]\!]_{\mathcal{S}_D}) \subseteq [\![post_{Sc}]\!]_{\mathcal{S}_D} .$$

The theorem only relates signature homomorphous ontologies $K_T \rhd_f K_D$ and mappings π_f which concretizes the templates' constraint rules in a way such that they comply with the target domain knowledge. Therefore we define $\mathcal{S}_D$ and π_f such that

$$\mathcal{S}_D \models R_D \text{ and } R_D \models \pi_f(CR) .$$

If $\mathcal{S}_D$ satisfies the rules of the target ontology, then it also satisfies the subset of rules which results from the signature homomorphous mapping:

$$\mathcal{S}_D \models R_D \quad \Rightarrow \quad \mathcal{S}_D \models \pi_f(R_T)$$

because by Definition 4.19 (Ontology Mapping) we know

$$\pi_f(R_T) \subseteq R_D .$$

Also, if both $R_D \models \pi_f(CR)$ and $\mathcal{S}_D \models R_D$, then $\mathcal{S} \models \pi_f(CR)$. Therefore

$$\mathcal{S}_D \models \pi_f(R_T) \wedge \mathcal{S}_D \models \pi_f(CR)$$

of course also holds. Up to here, we used a logical structure $\mathcal{S}_D$, including interpretations, based on the target ontology. Now, we construct a corresponding logical structure based on the template ontology: By Proposition 4.25 (Logical Structure Correspondence) we can create a $\mathcal{S}^{\rhd f}$ using f of the signature homomorphous mapping $K_T \rhd_f K_D$. Additionally, by Proposition 4.26 (Placeholder Mapping Correspondence), we construct a mapping $\pi_f^{\rhd f} : \mathcal{SP}_T \to \mathcal{SVC}_T$ within the template ontology such that

$$\mathcal{S}^{\rhd f} \models R_T \text{ and } \mathcal{S}^{\rhd f} \models \pi_f^{\rhd f}(CR) .$$

By Definition 4.11 (Correctness of Composition Templates), and because Sct is correct, we know

$$\begin{array}{rl} & \mathcal{S}^{\rhd f} \models R_T \;\text{ and }\; \mathcal{S}^{\rhd f} \models \pi_f^{\rhd f}(CR) \\ \text{such that} & [\![\, WT \,]\!]_{\mathcal{S}^{\rhd f}}^{\pi_f^{\rhd f}} \big([\![\, \pi_f^{\rhd f}(pre_{Sct}) \,]\!]_{\mathcal{S}^{\rhd f}}\big) \subseteq [\![\, \pi_f^{\rhd f}(post_{Sct}) \,]\!]_{\mathcal{S}^{\rhd f}} \;. \end{array}$$

Currenty, we talk about the semantics (and correctness) of the composition template. From Lemma 4.23 (Same Semantics of Workflow Templates and Instantiations), we know that a workflow template WT and a workflow $\pi(WT)$ have the same semantics, for $\pi : \mathcal{SP}_T \to \mathcal{SVC}_T$. We apply this to WT and $\pi_f^{\rhd f}(WT)$, and therefore

$$\begin{array}{rl} & \mathcal{S}^{\rhd f} \models R_T \;\text{ and }\; \mathcal{S}^{\rhd f} \models \pi_f^{\rhd f}(CR) \\ \text{such that} & [\![\, \pi_f^{\rhd f}(WT) \,]\!]_{\mathcal{S}^{\rhd f}} \big([\![\, \pi_f^{\rhd f}(pre_{Sct}) \,]\!]_{\mathcal{S}^{\rhd f}}\big) \subseteq [\![\, \pi_f^{\rhd f}(post_{Sct}) \,]\!]_{\mathcal{S}^{\rhd f}} \;. \end{array}$$

Now, we talk about the semantics of the service composition, containing the workflow $\pi_f^{\rhd f}(WT)$. Using Lemma 4.27 (Same Set of States), because $\mathcal{S}^{\rhd f}$ and $\mathcal{S}_D$ rely on the same interpretations, the same is true for our original structure $\mathcal{S}_D$ in the target ontology K_D:

$$\begin{array}{rl} & \mathcal{S}_D \models R_D \;\text{ and }\; \mathcal{S}_D \models \pi_f(CR) \\ \text{such that} & [\![\, \pi_f(WT) \,]\!]_{\mathcal{S}_D} \big([\![\, \pi_f(pre_{Sct}) \,]\!]_{\mathcal{S}_D}\big) \subseteq [\![\, \pi_f(post_{Sct}) \,]\!]_{\mathcal{S}_D} \;. \end{array}$$

It is therefore sufficient to show that $R_D \models \pi_f(CR)$, if the template is already proved to be correct, and $K_T \rhd_f K_D$ holds.

□

According to this theorem, it is now sufficient to (a) prove correctness of a template and (b) proof constraint rule compliance of the template with the target domain ontology of the instantiation.

Checking constraint rule compliance does not rely on the control and data flow of a composition. Therefore, even large compositions with complex control and data flows have to be proved to be correct only once, even manually if need be. Depending on the expressiveness of the pre- and postconditions of the services, and their dependencies encoded in the constraint rules, and the structure of these rules, the check for constraint rule compliance can even be executed by specialized ontology solvers like HermiT (Glimm et al., 2014) or Pellet (Sirin et al., 2007).

While this constraint check relies only on an encoding of the domain knowledge and the instantiated constraint rules (and therefore service descriptions), the next chapter introduces a method to verify service composition templates automatically.

Chapter 5

Automating Correctness Proofs using First-order Logic

The previous chapters discuss a technique to derive a correct service composition from a verified template, by checking side conditions instead of executing a complete verification. Depending on the structure of the constraint rules and the capabilities of the respective solver, the resulting check can be done within the domain of ontology reasoning, using standard tools. However, the premise is that the template is already proven to be correct, using the proof calculus. While it is possible to prove correctness of a template manually, it is of course more desirable to automate these checks. Especially in an on-the-fly context, full automation is an important goal, even if a template-based scenario relaxes timing constraints on template verification.

Section 5.1 elaborates on some of the topics of program verification based on logic. However, most of these works focus on verification of imperative programs and specific problems resulting from the use of mathematical expressions, e.g., using linear integer or floating point arithmetics, or from data structures, e.g., doing verification in the presence of pointers. The focus of our context is an integrating one. It includes not only encoding of control and data flow of workflows and the corresponding service descriptions, but also the context of the relevant domain knowledge, formalized using ontologies. It therefore is located in the program verification part of the three areas of this thesis, though rooted in the common logical foundation ($\rightarrow$ Figure 5.1).

This chapter presents a logical encoding of service composition templates and defines a correctness proof as a satisfiability problem. The use of standard SMT solvers makes inclusion of domain knowledge easy. The core part is the combination of this logical encoding, the result of the satisfiability check, and the construction of proof outlines as defined in Section 4.3 (Partial and Total Correctness). To this end, we define a logical encoding and then prove the equivalence of a satisfiability check result with the fact that a proof of correctness using the proof calculus can be constructed. In other words, if the satisfiability check undertaken by a solver succeeds, a formal proof can be constructed; If it does not succeed, there are two possible reasons:

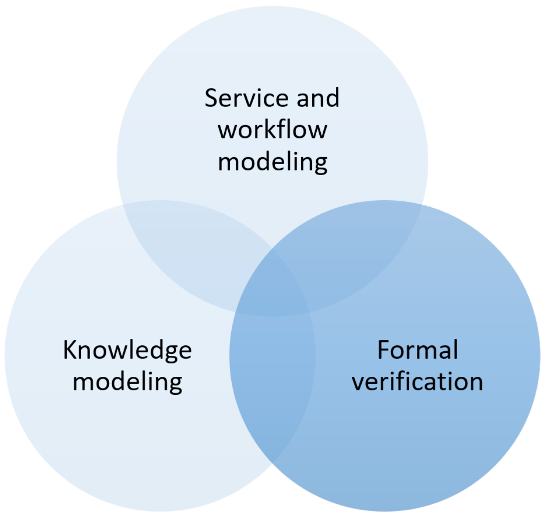

Figure 5.1: Joint research: Formal verification in relation to knowledg modeling and service and workflow modeling

- The solver provides a counterexample which shows a violation of the proof obligations. It consists of a variable assignment demonstrating a correct input, which leads – using the logical encoding of the composition – to an output which violates the requirements. This result indicates that such a proof is not possible.
- The solver cannot provide an answer. The reason for this is the use of undecidable theories, e.g., when trying to solve predicate logic formulas (→ Section 7.2).

Section 5.1 gives an overview of logic based verification and the treatment of loops. Section 5.2 defines the actual logical encoding of workflows, while Section 5.3 states a correspondence between correctness of a workflow and a satisfiability problem, including the proofs. Finally, Section 5.4 discusses first steps of how to integrate domain knowledge into existing techniques to automatically discover loop invariants and termination arguments.

5.1 Related Work and the Treatment of Loops

Generally, *model checking* is a way to prove or disprove the presence of properties in a system model (Clarke and Emerson, 1982). Its basic idea is to explore the state space of a system model and check whether the desired property holds or is violated. If this is done explicitly, the *state explosion problem* is immediately obvious: The number of states of a program grows exponentially in its size with the number of variables. *Symbolic* model checking avoids an explicit enumeration of a state space and aims at compact representations which can be checked more efficiently (Burch et al., 1992). Symbolic representations both of system properties and the system model itself can be achieved in terms of logical formulas, which are often encoded as *binary decision diagrams* (BDDs, Lee, 1959, Akers, 1978). The efficiency of BDD based model checking algorithms depends on topics like BDD normalization and appropriate variable ordering. Both have a huge impact on the size of the BDD representation of a model: Normalization (including efficient representation) depends on the order of variables, which in turn highly depends on the formula at hand; an order suitable for efficient representation is not easy to compute automatically (Biere et al., 1999).

Apart from using BDDs for compact formula representation, it is also possible to use logical encodings of model checking problems and utilize satisfiability solvers directly. When we encode states and state transitions by means of assertions on variable values, it is not immediately clear how we should represent loops. The execution of a loop body changes the system state, but as it is not a priori known how often a loop will be executed, it is not possible to replace it with linear constructs. There are two approaches to address this issue: One is to execute the loop up to a fixed number of times by *unrolling* it, the other is to summarize its behavior using a loop *invariant*.

The basic idea of *bounded model checking* is to artificially restrict the (potentially) infinite state space as induced by loops by restricting the number of loop executions. To this end, the loop is *unrolled*, that is, the loop is replaced by a repeating sequence of the loop's body, up to a fixed bound k. This way, the original behavior of the loop is exactly represented as long as the loop is executed at most k times (Biere et al., 1999). As a result, if some property happens to hold (or to be violated) only after k unrollings, this is naturally not detectable. With CBMC, or *C bounded model checking*, there exists an approach where bounded model checking is applied to C programs, using a logical encoding of states and state transitions (Clarke et al., 2004). Here, possible program traces are encoded as logical formula, and reachability of an error state – that is, an error location in terms of a violated assert statement – as a satisfiability problem. This problem can be solved by a satisfiability solver.

While CBMC is not only bounded in terms of loop unrolling, it is also restricted in terms of data type representation, namely the size of arrays or lists. This is due to their logical encoding using Boolean formulas. Armando et al. lift some of these restrictions by applying the the CBMC approach to an SMT encoding instead of SAT. This way, they make use of the theories of linear arithmetics, lists, and bitvectors to lift the restriction of fixed array lengths (Armando et al., 2009). Milicevic and Kugler use the theory of lists to address the issue of a fixed bound k of bounded model checking. They encode program states using list variables. As SMT lists are potentially unbounded, this enables implicit loop unrolling without a fixed bound, covering potentially infinite program traces (Milicevic and Kugler, 2011). If the solver terminates, the safety property in question is either disproved or proved without any bound.

The classical representation of loops by *loop invariants*, as used in the theoretical part of this thesis, summarizes the overall behavior of a loop, with no regard to a specific number of executions (Hoare, 1969). While invariants are a convenient representation of loops in calculi, they are also used in practice. The *Java Markup Language* (JML), for example, uses invariants to annotate not only loops, but complete Java methods (Leavens et al., 1999). Invariants are used in verification techniques that build upon JML, like ESC/Java (Cok and Kiniry, 2005, Flanagan et al., 2002). While loop invariants do not restrict the number of loop executions, they have their own disadvantage: They are usually required to be given explicitly, that is, to be defined manually.

However, automatically *deriving* invariants from program code is not a new research topic by itself. Already Karr as well as Cousot and Halbwachs developed techniques to find invariant expressions of loops. They take the form of linear inequalities of variable expressions and a constant (Cousot and Halbwachs, 1978, Karr, 1976). In practice, with tools like InvGen it is possible to define templates or patterns of loop invariants. These templates are used to find linear inequalities and Boolean combinations of them automatically (Gupta and Rybalchenko, 2009). Sharma et al. extend the use of Boolean combinations to Boolean disjuncts or disjunctive invariants (Sharma et al., 2011). Beyer et al. synthesize invariants expressed in SMT formulas and combine the theory of linear arithmetics (to express linear inequalities) with uninterpreted functions (Beyer et al., 2007). Srivastava and Gulwani also find invariants based on invariant patterns, including quantified expressions. To this end, they find solutions to the variable part of an invariant pattern by fix point computation (Srivastava and Gulwani, 2009).

Instead of inspecting the loop body to compute inequalities, invariants can also be inferred based on the postcondition of a loop. To this end, this postcondition is mutated to generate invariant candidates, which are then checked for actual invariance (Furia and Meyer, 2010). Heuristics to generate these mutations include replacing constants of a postcondition with variables (constant relaxation), replacing two occurrences of a variable with two different variables (uncoupling), or removing parts of the postcondition completely (term dropping) (Furia and Meyer, 2010, Gries, 1981, Meyer, 1980)

The tool DynaMate combines these approaches. It is built upon ESC/Java2 and InvGen and finds invariants both based on patterns and on mutating postconditions. Invariant candidates are then tested and falsified ones eliminated. The remaining candidates are verified whether or not they are invariant and actually useful for the current verification task. Detectable invariants are not only linear inequalities, but also quantified expressions, as long as these can be generated from the postcondition of a loop (Galeotti et al., 2014).

With no regard to the exact approach to derive loop invariants there is always one potential problem to be considered: The *assertion inference paradox*. The assertion inference paradox is a potential vicious circle: On the one hand, we need the loop invariant to prove the correctness of the program with respect to its postcondition. This includes the loop being part of the program's behavior. On the other hand, if we *derive* the invariant from the program, proving correctness using this invariant may carry no meaning due to this cyclic dependency (Furia and Meyer, 2010, p. 278). This paradox is not *necessarily* a problem, but should be considered by every approach of invariant generation. The criterion to avoid this paradox is the question of which information is present to begin with: If an invariant is derived, but pre- and postcondition of the overall program are *given* in a verification approach, the complete proof is still valid (Furia and Meyer, 2010).

In this thesis, we assume that invariants are given, with no additional restriction apart from that they are expressible using the terms built on the vocabulary of the domain knowledge (→ Definition 2.9, First-order Formulas). However, in Section 5.4 (Deriving Invariants and Termination Functions) we propose techniques to leverage the inference of loop invariant candidates by utilizing the existing domain knowledge.

5.2 First-order Logic Encodings

This section defines a logical encoding of workflows and formulates proof obligations to define a logical formula representing the correctness of a workflow template with respect to its pre- and postcondition. A theorem establishes a correspondence between this formula being a tautology and the provability of the correctness of the workflow template.

5.2.1 Preliminaries

The overall knowledge of a certain domain is formalized using ontologies, whose semantics can be described with description logics, which are subsets of first-order logic. Additionally, our service descriptions use first-order logical expressions based on the vocabulary derived from the underlying ontology, and our definitions of correctness as well as the semantics of our workflow definition are based on logic. With pre- and postconditions encoded using logical formulas, we encode the control and data flow with logical formulas as well. To this end, we need to identify positions within the control flow of a service composition (or template). We use *labels* to identify this position. These labels are already part of Definition 4.5 (Workflow Template):

$$\begin{aligned} WT ::= {} & [l]\ \mathit{skip} \\ \mid {} & [l]\ u := t \\ \mid {} & WT_1;\ WT_2 \\ \mid {} & [l]\ (u_{j+1}, \ldots, u_k) := S(i_1, \ldots, i_j) \\ \mid {} & [l]\ \texttt{if}\ B\ \texttt{then}\ WT_1\ \texttt{else}\ WT_2\ \texttt{fi} \\ \mid {} & [l]\ \texttt{while}\ B\ \texttt{do}\ WT_1\ \texttt{od} \\ \mid {} & [l]\ \texttt{foreach}\ a \in A\ \texttt{do}\ WT_1\ \texttt{od} \end{aligned}$$

Here, all labels l are supposed to be unique, and we define a final label $[end]$. This way, every statement has exactly one *next label*: Either the label of the next statement in a sequence, the next label of its parent (in case of conditional or loop statements), or the end label. By labeling a statement, we are able to exactly determine the current position in a workflow template the same way as if using a program counter variable. We write $[l]\ WT\ [n]$ to refer to a workflow template WT which is labeled with l and which has n as the next label.

We make use of this *next* label when we encode statements as logical formulas, esp. as indices for variables. We use Var to denote the set of variables of a complete workflow template ($\rightarrow$ Definition 2.7). For every variable in Var, we also assume an indexed version, and define a corresponding set of variables with a given index.

Definition 5.1 (Indexed Variables)**.** For a set Var of variables, and an index i, we define a set Var_i of indexed variables:

$$Var_i := \{x_i \mid x \in Var\} \ .$$

We also make use of syntactic replacement of variable names in expressions, e.g., $p[x := x_i]$ refers to the expression p where every occurrence of x is replaced by x_i. Using labels as variable indices, we create different versions of each variable corresponding to the workflow position, similar to the use of versioned variables in *static single assignment* form (SSA form, Cytron et al., 1991). In addition to the syntactic replacement of one variable as in $p[x := z]$, we define a shorthand notation to replace all variables in an expression or assertion by their indexed versions.

Definition 5.2 (Variable Replacement for all Variables in Expressions)**.** Let p be a logical expression with $\mathit{free}(p) \subseteq Var$. We define p_i as p with every variable replaced by its indexed counterpart:

$$p_i := p[x := x_i]$$

with $x \in Var$ and $x_i \in Var_i$ for every variable $x \in \mathit{free}(p)$.

We will use this shorthand particularly in assignment expressions and conditional expressions. As the shorthand form of indexed expressions replaces only *free* variables in p, *quantified* variables are never indexed.

Example 5.1 (No Replacement of Bound Variables)**.** *Let* $p := \forall x : x \neq y$. *Then* $\mathit{free}(p) = \{y\}$ *and* $p_i = (\forall x : x \neq y_i)$.

5.2.2 Domain Knowledge and Services

A formal knowledge base as given in Definition 2.3 with its predicate logic semantics is already a logical encoding of itself.

Definition 5.3 (Knowledge Base Encoding)**.** Let $K = (C, P, \sqsubseteq, R)$ be a knowledge base. We define a locical encoding φ as follows: We use concepts C as unary predicates, roles P as binary predicates, and rules R as all-quantified logical expressions. We encode the type hierarchy given by $C_1 \sqsubseteq C_2$ as $\forall x : C_1(x) \Rightarrow C_2(x)$.

To encode typing of predicates, we will add the use of type predicates to every use of a role predicate. Then, the typed predicate

$$\textsc{hasRating} : \textsc{Restaurant} \times \textsc{Rating}$$

used as in

$$\mathit{hasRating}(x, y)$$

translates to

$$\mathit{hasRating}(x, y) \wedge \mathit{Restaurant}(x) \wedge \mathit{Rating}(y) \ .$$

For easier reading, we will omit this additional notation. Chapter 7 discusses its translation into SMT-LIB, a standard language for satisfiability solver input. Logical encoding of services is straightforward, too, as their pre- and postcondition as the relevant parts of a service are already logical expressions, including the type information from the signature. Therefore the same technique is applicable.

5.2.3 Control Flow

In the following, we inductively define the logical encoding of statements, relating variable versions of the state *before* the execution of the statement to the state *after* the execution. We refer to these states using the *labels* from Definition 3.3, especially the *next* label of a workflow (which may be the closing *end* label). In summary, a logical encoding φ_W of a statement $[l]\ W\ [n]$ with label l and next label n uses the labels l and n as indices in the encoding of variables.

Service calls make use of the pre- and postconditions of service descriptions, which are expressions using input and output variables as denoted in the service description. As in the previous chapters, we assume one input (named i) and one output (named o) for services for easier reading. To make use of pre- and postconditions, the variables have to be replaced by the actual variables with which a service is called. We denote this by syntactic replacement, e.g., $\mathit{post}[i := x, o := y]$. Pre- and postconditions of services refer to complex expressions ($\mathit{pre}, \mathit{post} \in \Phi_K$), but range only over variables occurring in the signature of the service.

To represent loops, we use a logical encoding of loop invariants. However, while all invariants of a loop reflect its behavior, not all of them can necessarily be used to prove correctness of a workflow template. The reason is the *context*, or the set of preceding statements, of a loop: The facts that hold true for these preceding statements according to the proof rules ($\rightarrow$ Def. 4.12, Parameterized Proof Calculus for Workflow Templates) must *imply* the invariant which is actually part of the proof outline and which (after the loop) implies the following facts. Eventually, this chain of implications leads to the postcondition of the workflow template, and is implemented by the *rule of consequence* (rule 9). Therefore, the invariant used to encode a loop cannot be arbitrary if the encoding should correspond to the correctness of the workflow. Instead, it must be *implied* by the encoding of the previous statements. To take this relation into account, we need to show that an invariant holds at the beginning of a loop. If we want to prove the (partial) workflow in Figure 5.2, this is only possible if the invariant at the `while` loop can be implied from the facts that hold true in front of the `if` statement and the negated `if` condition.

```
  [1]  $y := S_1(x);$
  [2]  **if** $B_1(y)$ **then**
  [3]     $z := S_2(y)$
       **else**
  [4]     **while** $\neg B_2(x, y)$ **do**
  [5]        $y, z := S_3(y)$
          **od**
       **fi**
[end]
```

FIGURE 5.2: Workflow with a conditional loop statement

This is not necessarily true for all possible invariants of the loop. To supplement the logical encoding of the actual workflow statements, we introduce *marker variables* for every position in a workflow. This way, we are able to “track” the control flow which is used for a variable assignment of the logical encoding.

Definition 5.4 (Position Marker Variables)**.** For every label l used in a workflow template, or workflow, we define a *marker variable* m_l of type Bool, and the set of marker variables of a workflow as

$$M = \{m_l \mid l \text{ is a label of the workflow (template)}\}$$

and $M \cap \mathit{Var} = \emptyset$, that is, M is disjoint with the regular set of variables.

Marker variables are used to explicitly mark whether or not a statement is “active” for a current state, that is, variable assignment. This way, marker variables indicate whether for a given variable assignment the *then* or *else* branch of a conditional statement is taken, and whether or not the current statement is taken at all. In other words, they mark the actually used control flow for a given variable assignment.

Example 5.2 (Marking the `else` Branch)**.** *For the workflow in Figure 5.2, we have the labels l_1 to l_5 and an additional label l_{end}. We define the logical encoding of workflow statements such that if for a given variable assignment the condition $B_1(y)$ is false, then the marker for label l_3 will be false, too, but the marker for label l_4 will be true.*

Marker variables are “routed” through the workflow as part of the logical encoding of composite statements. Their initial value will be set by the verification conditions which encode the overall correctness, as we will see later on. This way, we will be able to identify whether or not we actually need to have an invariant to hold true.

For easier reading, we also occasionally use $\texttt{ite}(B, WT_1, WT_2)$ as shorthand to denote $\texttt{if}\ B\ \texttt{then}\ WT_1\ \texttt{else}\ WT_2\ \texttt{fi}$. Please note, that a workflow can be treated as a workflow template without service placeholders and constraint rules.

Definition 5.5 (Logical Encoding of Workflow Templates)**.** Let WT be a workflow template. We define the logical representation of $[l]\ WT\ [n]$, $\varphi^{l,n}_{WT}$, inductively on its structure, using its labels as indices of the variables. In the encodings, **inv** denotes an invariant of the respective loop. For service calls, we use different encodings for total and partial correctness.

Statement	**Example**	**Encoding**
E	(not applicable)	$\varphi_E := \top$
Skip	$[l]\ \mathit{skip}\ [n]$	$\varphi^{l,n}_{\mathit{skip}} := \bigwedge_{x \in \mathit{Var}} x_l = x_n$
Assignment	$[l]\ u := t\ [n]$	$\varphi^{l,n}_{u:=t} := u_n = t_l \wedge \bigwedge_{x \in \mathit{Var} \setminus \{u\}} x_n = x_l$
Service call (part. corr.)	$[l]\ u := S(v)\ [n]$	$\varphi^{l,n}_{u:=S(v)} := \big(\mathit{post}[i := v, o := u]_n \wedge \bigwedge_{x \in \mathit{Var} \setminus \{u\}} x_n = x_l\big)$
Service call (total corr.)	$[l]\ u := S(v)\ [n]$	$\varphi^{l,n}_{u:=S(v)} := \mathit{pre}[i := v]_l \Rightarrow \big(\mathit{post}[i := v, o := u]_n \wedge \bigwedge_{x \in \mathit{Var} \setminus \{u\}} x_n = x_l\big)$
While loop	$[l]\ \texttt{while}\ B\ \texttt{do}$ $[k]\ WT\ \texttt{od}\ [n]$	$\varphi^{l,n}_{\texttt{while}\ B\ \texttt{do}\ WT\ \texttt{od}} := \mathbf{inv}_l \wedge (\mathbf{inv}_n \wedge \neg B_n) \wedge \bigwedge_{x \in \mathit{Var} \setminus \mathit{change}(WT)} x_l = x_n$
Foreach loop	$[l]\ \texttt{foreach}\ a \in A\ \texttt{do}$ $[k]\ WT\ \texttt{od}\ [n]$	$\varphi^{l,n}_{\texttt{foreach}\ a \in A\ \texttt{do}\ WT\ \texttt{od}} := \mathbf{inv}_l \wedge (\mathbf{inv}_n \wedge A_n = \emptyset) \wedge \bigwedge_{x \in \mathit{Var} \setminus \mathit{change}(WT)} x_l = x_n$
Sequence	$[l]\ WT_1; [k]\ WT_2\ [n]$	$\varphi^{l,n}_{WT_1;WT_2} := \varphi^{l,k}_{WT_1} \wedge \varphi^{k,n}_{WT_2} \wedge (m_k = m_l)$
Conditional	$[l]\ \texttt{if}\ B\ \texttt{then}\ [k]\ WT_1$ $\texttt{else}\ [h]\ WT_2\ \texttt{fi}\ [n]$	$\varphi^{l,n}_{\texttt{ite}(B, WT_1, WT_2)} := \big(m_l = m_k \wedge \neg m_h \wedge B_l \wedge \varphi^{k,n}_{WT_1} \wedge \bigwedge_{x \in \mathit{Var}} x_l = x_k\big) \vee \big(m_l = m_h \wedge \neg m_k \wedge \neg B_l \wedge \varphi^{h,n}_{WT_2} \wedge \bigwedge_{x \in \mathit{Var}} x_l = x_h\big)$

In the conditional encoding, each branch inherits its activity information (by the marker variable m) from the overall statement and at the same time it explicitly "deactivates" the other branch.

Example 5.3 (Workflow Template Encoding)**.** *We encode the following part of a workflow template, using numbers as labels:*

```
[1]  if $P(a)$ then
[2]      $x := a$;
[3]      $y := b$;
     else
[4]      $x := b$;
[5]      $y := a$;
     fi
[6]
```

Every statement has one label and one next label, e.g., the next label of $x := a$ is 3, the next label of $y := b$ is 6 (which is also the next label after the **if** *clause). If we apply Definition 5.5, we get the following formula:*

$$\begin{aligned}
&\Big(\big(P(a_1) \wedge m_1 = m_2 \wedge \neg m_4 \\
&\quad \wedge a_2 = a_1 \wedge b_2 = b_1 \wedge x_2 = x_1 \wedge y_2 = x_1 \\
&\quad \wedge (x_3 = a_2 \wedge a_3 = a_2 \wedge b_3 = b_2 \wedge y_3 = y_2) \\
&\quad \wedge (y_6 = b_3 \wedge a_6 = a_3 \wedge b_6 = b_3 \wedge x_6 = x_3)\big) \\
\vee\ &\big(\neg P(a_1) \wedge m_1 = m_4 \wedge \neg m_2 \\
&\quad \wedge a_4 = a_1 \wedge b_4 = b_1 \wedge x_4 = x_1 \wedge y_4 = y_1 \\
&\quad \wedge (x_5 = b_4 \wedge a_5 = a_4 \wedge b_5 = b_4 \wedge y_5 = y_4) \\
&\quad \wedge (y_6 = a_5 \wedge a_6 = a_5 \wedge b_6 = b_5 \wedge x_6 = x_5)\big)\Big)
\end{aligned}$$

Variables using label 1 as index represent variables "before" the workflow starts, label 6 "after" the workflow. The marker variables m_1, m_2, m_4 represent which branch of the **if** *clause is active for a given variable assignment.*

Based on logical encodings of workflows, we proceed to the definition of a correspondence between tautology of a formula and correctness of a workflow template.

5.3 Correspondence of Correctness and SAT Problems

A logical encoding is not an automated application of proof rules according to the proof calculus of Section 4.4. It is therefore necessary to define, and prove, a correspondence between a satisfiability problem based on a logical encoding of a template and the formal proof calculus.

5.3.1 Correspondence Theorem

Instead of creating a proof outline manually, we use an SMT solver to determine whether or not a workflow template is correct. To do so, we need a correspondence between the provability of a correct service composition template (→ Definition 4.14), and the validity of a formula which consists of the logical encoding of its workflow (→ Definition 5.5). The theorem distinguishes between partial and total correctness of a template, depending on whether or not termination is part of the proof. In practice, this depends on whether or not a termination function of the `while` loop is present. To formalize this, we define the set of loops of a workflow as well as their corresponding invariants and termination functions.

Definition 5.6 (Loops, Invariants, and Termination Functions)**.** Let WT be the workflow of a service composition template. We address each `foreach` and `while` loop using the keyword *loop* indexed with the label of the loop. We denote the *invariant* of $loop_l$ as

$$\mathbf{inv}_l \in \Phi_K$$

and its *termination function* as an expression

$$\mathbf{t}_l \in \Phi_K \text{ with } type(\mathbf{t}_l) = Integer\ .$$

We define the set of all loops of WT as

$$L := \left\{ loop_l \mid loop_l \text{ is a loop in } WT \text{ labeled } [l] \right\}$$

and the set of *top level loops* as

$$L_{WT} := \begin{cases} \emptyset & \text{if } WT = E, skip, \text{assignment, or service call} \\ \{\mathbf{inv}_l\} & \text{if } WT = [l]\ loop\ B\ \texttt{do}\ [k]\ W_1\ \texttt{od}\ [n] \\ L_{W_1} \cup L_{W_2} & \text{if } WT = W_1; W_2 \text{ or } \texttt{if}\ B\ \texttt{then}\ W_1\ \texttt{else}\ W_2\ \texttt{fi} \end{cases}$$

Please note that the definition of top level loops does not contain nested loops.

As a loop invariant reflects the behavior of the loop, its encoding is a valid representation to treat the loop as a black box, that is, to ignore the encoding of the actual loop body. However, not every loop invariant can be used within a proof of correctness, as loops are no black boxes in proofs. Instead, a loop invariant must be implied by the preceding statements, hence the *rule of consequence* must be applicable (→ Def. 4.12, Parameterized Proof Calculus for Workflow Templates, rule 9). This is not necessarily true for every possible loop invariant. Therefore, it is not sufficient to use an arbitrary invariant to encode a loop, but an invariant that is implied by the encoding of the preceding statements, or its *context.* One option to do this is to explicitly encode the sequence that precedes a given loop invariant, in addition to the encoding of the overall workflow. This approach has a major drawback: In the proofs of the correspondence theorem defined below, all possible contexts of a loop have to be considered.

Instead, we utilize the existing encoding of the overall workflow by using the marker variables. Marker variables are by definition only valid, if a specific path in a workflow is "active", that is, the variable assignments are such that the corresponding branch of a conditional statement is taken. Whenever it is the case that a path to an invariant is active, we require that the overall encoding implies the invariant.

Definition 5.7 (Verification Condition)**.** Let $[l]$ WT $[n]$ be a workflow template and $p, q \in \Phi_K$. We define the *verification condition* of the workflow template regarding p and q as

$$VC(WT, p, q) := \quad \big(m_l \wedge p_l \wedge \varphi^{l,n}_{WT}\big) \Rightarrow \big(q_n \wedge \bigwedge_{loop_k \in L_{WT}} (m_k \Rightarrow \mathbf{inv}_k)\big) \ .$$

In other words, given a precondition p and a workflow encoding, the postcondition q is implied. Additionally, for every top level loop in the workflow (that is, without nested loops), the invariant actually used to encode the loop must be implied at the loop position, marked by m_k.

As loop encodings are black bloxes, errors in the loop body cannot be detected if we create a verification condition for the overall workflow template of a service composition. Therefore, we have to prove that a given invariant really *is* an invariant of a loop. We do this by creating verification conditions not only for the workflow template of a service composition, but for every loop of the workflow as well.

Definition 5.8 (Proof Obligation for Partial Correctness)**.** Let (Sct^{Desc}, WT, CR) be a service composition template with a workflow labeled with $[l]$ WT $[n]$, precondition *pre*, and postcondition *post*. We define the *proof obligation for partial correctness* for the service composition template as

$$\begin{aligned} \textsc{Req} := \quad & VC(WT, pre, post) \wedge \\ & \bigwedge_{loop_i \in L} VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \end{aligned}$$

with $[i]$ *loop* B `do` $[j]$ W `od` $[k]$.

Please note, that in this proof obligation *all* loops are covered, including nested loops. In addition to the proof obligation for partial correctness, for total correctness we have to show termination for every loop.

Definition 5.9 (Proof Obligations for Total Correctness)**.** Let (Sct^{Desc}, WT, CR) be a service composition template with loops L. Then, for $loop_i \in L$ with labels $[i]$ *loop* B `do`

$[j]\ W$ `od` $[k]$ (and $B := A \neq \emptyset$ in case of `foreach` loops),

$$\begin{aligned} \text{T-PROG}_i :=&\quad \big(\mathbf{inv}_j \wedge B_j \wedge \mathbf{t}_j = z \wedge \varphi_W^{j,k}\big) \Rightarrow \mathbf{t}_k < z \\ \text{T-BOUND}_i :=&\quad \mathbf{inv} \Rightarrow \mathbf{t} \geq 0 \end{aligned}$$

with $\mathbf{t} \in \Phi_K$ the termination function expression, z a fresh variable and $type(\mathbf{t}) = type(z) = Integer$ denote the loop-related proof obligations.

Please note that the encodings for partial and total correctness differ not only by the encoding of loop progress and termination, but also in the encodings of service calls.

With the requirements proof obligation and the loop termination proof obligations we can formalize a theorem to relate provable correctness in the proof calculus with the tautology of a logical formula.

Theorem 5.10 (Provability corresponds with Tautology)**.** *Let* (Sct^{Desc}, WT, CR) *be a service composition template over a domain ontology* K*, with* $[l]\ WT\ [n]$ *the labeled workflow template of* Sct *and* L *with* $loop_i \in L$ *labeled* $[i]$ *loop* B `do` $[j]\ W$ `od` $[k]$ *(and* $B := A \neq \emptyset$ *in case of* `foreach` *loops) the loops of* WT*.*

For partial correctness, the following holds for two expressions p, q*:*

$$\begin{aligned} &\vdash_K^{CR} \{p\}\, Sct\, \{q\} \\ \Leftrightarrow \quad &\models_K^{CR} \text{REQ}\ . \end{aligned}$$

For total correctness, the loops have to terminate:

$$\begin{aligned} &\vdash_K^{CR} \{p\}\, Sct\, \{q\} \\ \Leftrightarrow \quad &\models_K^{CR} \Big(\text{REQ} \wedge \bigwedge_{loop_l \in L} \big(\text{T-PROG}_l \wedge \text{T-BOUND}_l\big)\Big) \end{aligned}$$

With this theorem, we can automatically create a logical representation of a given workflow and use a satisfiability solver to check for correctness. As the correctness of a service composition is equivalent to the *tautology* of the logical encoding, but solvers check for *satisfiability*, in practice the encoding is negated and checked for contradiction, as $\big(\models P\big) \Leftrightarrow \big(\neg P \equiv \bot\big)$. Chapter 7 discusses a prototypical implementation and a translation into a solver input language, including the details on encoding a tautology check as a contradiction check.

5.3.2 Proofs

To prove Theorem 5.10 we have to prove two directions: At first, we show that whenever we have a provably correct service composition template with respect to some pre- and postcondition, then the logical encoding which results from the theorem is a tautology ("⇒"). At second, we show that whenever we have a logical encoding according to the theorem, and the resulting formula is a tautology, then we are able to construct a proof outline such that we can prove that the workflow (and therefore the service composition template) is correct (“$\Leftarrow$”).

Proof. Let (Sct^{Desc}, WT, CR) be a service composition template with workflow $[l]\ WT\ [n]$ over a knowledge base K. Let $p, q \in \Phi_K^{sp}$. We show that if Sct is correct regarding p and q, the resulting logical encoding is a tautology, that is

$$\Big(\vdash_K^{CR} \{p\}\ WT\,\{q\}\Big) \Rightarrow \Big(\models_K^{CR} \textsc{Req}\Big)\ .$$

for partial correctness, and

$$\Big(\vdash_K^{CR} \{p\}\ WT\,\{q\}\Big) \Rightarrow \Big(\models_K^{CR} \textsc{Req} \wedge \bigwedge_{loop_l \in L} (\text{T-}\textsc{Prog}_l \wedge \text{T-}\textsc{Bound}_l)\Big)$$

for total correctness. We prove this by induction, using skip, assignment, service call, and loops as induction base cases. Therefore, both L and L_{WT} are empty for the base cases except the loops.

Skip We know: $\vdash_K^{CR} \{p\}\ skip\,\{p\}$ with labels $[l]\ skip\ [n]$.

$$\begin{array}{ll} & m_l \wedge p_l \wedge \varphi_{skip}^{l,n} \\ \Leftrightarrow & \text{by Def. 5.5 (Skip)} \\ & m_l \wedge p_l \wedge \bigwedge_{x \in Var} x_l = x_n \\ \Rightarrow & p_n \end{array}$$

Assignment We know: $\vdash_K^{CR} \{p[u := t]\}\, u := t\,\{p\}$ with labels $[l]\ u := t\ [n]$, where $u \in Var$ and $t \in \Phi_K$ with $type(u) = type(t)$.

$$\begin{array}{ll} & m_l \wedge p[u := t]_l \wedge \varphi_{u:=t}^{l,n} \\ \Leftrightarrow & \text{by Def. 5.5 (Assignment)} \\ & m_l \wedge p[u := t]_l \wedge u_n = t_l \wedge \bigwedge_{x \in Var \setminus \{u\}} x_n = x_l \\ \Rightarrow & \text{by Def. of Substitution and } \mathcal{I}(u_n) = \mathcal{I}(t_l) \\ & p_n \end{array}$$

Service call (partial correctness)

We know: $\vdash_K^{CR} \{\forall w : post[i := v, o := w] \Rightarrow q[u := w]\}\, u := S(v)\, \{q\}$ with labels $[l]\ u := S(v)\ [n]$, where i is an input and o an output variable in the signature, *pre* the precondition, and *post* the postcondition of S. Reminder: According to Definition 5.2, quantified variables are never indexed.

$$\begin{array}{ll}
& m_l \wedge \big(\forall w : post[i := v, o := w] \Rightarrow q[u := w]\big)_l \wedge \varphi^{l,n}_{u:=S(v)} \\
\Leftrightarrow & \text{by Def. 5.5 (Service Call)} \\
& m_l \wedge \big(\forall w : post[i := v, o := w] \Rightarrow q[u := w]\big)_l \\
& \wedge \big(post[i := v, o := u]_n \wedge \bigwedge\limits_{x \in Var \setminus \{u\}} x_n = x_l\big) \\
\Rightarrow & q_n
\end{array}$$

Service call (total correctness)

We know: $\vdash_K^{CR} \{pre[i := v] \wedge \forall w : post[i := v, o := w] \Rightarrow q[u := w]\}\, u := S(v)\, \{q\}$ with labels $[l]\ u := S(v)\ [n]$, where i is an input and o an output variable in the signature, *pre* the precondition, and *post* the postcondition of S. Reminder: According to Definition 5.2, quantified variables are never indexed.

$$\begin{array}{ll}
& m_l \wedge \big(pre[i := v] \wedge \forall w : post[i := v, o := w] \Rightarrow q[u := w]\big)_l \wedge \varphi^{l,n}_{u:=S(v)} \\
\Leftrightarrow & \text{by Def. 5.5 (Service Call)} \\
& m_l \wedge \big(pre[i := v] \wedge \forall w : post[i := v, o := w] \Rightarrow q[u := w]\big)_l \\
& \wedge \big(pre[i := v]_l \Rightarrow post[i := v, o := u]_n \wedge \bigwedge\limits_{x \in Var \setminus \{u\}} x_n = x_l\big) \\
\Rightarrow & \text{modus ponens} \\
& m_l \wedge \big(pre[i := v] \wedge \forall w : post[i := v, o := w] \Rightarrow q[u := w]\big)_l \\
& \wedge\, post[i := v, o := u]_n \wedge \bigwedge\limits_{x \in Var \setminus \{u\}} x_n = x_l \\
\Rightarrow & q_n
\end{array}$$

While loop Let B be the loop condition, WT be a workflow template, and φ_{WT} its logical representation. Let l, k, n be labels as follows: $[l]$ `while` B `do` $[k]$ WT `od` $[n]$. The premise is $\vdash_K^{CR} \{\mathbf{inv}\}\,\texttt{while}\ B\ \texttt{do}\ WT\ \texttt{od}\{\mathbf{inv} \wedge \neg B\}$, with $\mathbf{inv} \in L$ being an *invariant* of the loop. For partial correctness, it is sufficient to show $\models_K^{CR}$ Req. For easier reading, we write $X = \mathit{Var} \setminus \mathit{change}(WT)$. For a given loop $[h]$ *loop* B `do` $[i]$ W `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-Prog}_h \wedge \text{T-Bound}_h$ for total correctness.

$$
\begin{aligned}
& \Big(\big(m_l \wedge \mathbf{inv}_l \wedge \varphi^{l,n}_{\texttt{while}\ B\ \texttt{do}\ WT\ \texttt{od}}\big) \Rightarrow \big(\mathbf{inv}_n \wedge \neg B_n \wedge (m_l \Rightarrow \mathbf{inv}_l)\big)\Big) \wedge \\
& \bigwedge_{loop_j \in L} VC_j \\
\Leftrightarrow \quad & \text{by Def. 5.5 (While)} \\
& \Big(\big(m_l \wedge \mathbf{inv}_l \wedge (\mathbf{inv}_l \wedge \mathbf{inv}_n \wedge \neg B_n \wedge \bigwedge_{x \in X} x_l = x_n)\big) \\
& \Rightarrow \big(\mathbf{inv}_n \wedge \neg B_n \wedge (m_l \Rightarrow \mathbf{inv}_l)\big)\Big) \wedge \\
& \bigwedge_{loop_j \in L} VC_j \\
\Leftrightarrow \quad & \text{by } \big(A \Rightarrow B \wedge C\big) \Leftrightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big) \\
& \Big(\big(m_l \wedge \mathbf{inv}_l \wedge (\mathbf{inv}_l \wedge \mathbf{inv}_n \wedge \neg B_n \wedge \bigwedge_{x \in X} x_l = x_n)\big) \Rightarrow \big(\mathbf{inv}_n \wedge \neg B_n\big)\Big) \wedge \\
& \Big(\big(m_l \wedge \mathbf{inv}_l \wedge (\mathbf{inv}_l \wedge \mathbf{inv}_n \wedge \neg B_n \wedge \bigwedge_{x \in X} x_l = x_n)\big) \Rightarrow \big(m_l \Rightarrow \mathbf{inv}_l\big)\Big) \wedge \\
& \bigwedge_{loop_j \in L} VC_j \\
\Rightarrow \quad & \text{by } (A \Rightarrow A) = \top \text{ and } \big((A \wedge B) \Rightarrow (A \Rightarrow B)\big) = \top \\
& \top \wedge \top \wedge \bigwedge_{loop_j \in L} VC_j \\
\Leftrightarrow \quad & \bigwedge_{loop_j \in L} VC_j
\end{aligned}
$$

Now, by induction, overall correctness depends on the correctness of inner loops. For total correctness, we also have to show termination of the loop. While the premise is the same, from the proof rule for total correctness we can also conclude that $\vdash_K^{CR} \{\mathbf{inv} \wedge B \wedge \mathbf{t} = z\}\ WT\ \{\mathbf{t} < z\}$ and $\models_K^{CR} \mathbf{inv} \Rightarrow \mathbf{t} \geq 0$ hold. Termination occurs in Theorem 5.10 by T-Prog and T-Bound.

$\big(\models_K^{CR} \text{T-Bound}\big) \Leftrightarrow \big(\models_K^{CR} \mathbf{inv} \Rightarrow \mathbf{t} \geq 0\big)$ follows directly from the premise.

$\big(\models_K^{CR} \text{T-Prog}\big) \Leftrightarrow \big(\models_K^{CR} \big(\mathbf{inv}_k \wedge B_k \wedge \mathbf{t}_k = z \wedge \varphi^{k,n}_{WT}\big) \Rightarrow \mathbf{t}_n < z\big)$ follows from the premise and the induction hypothesis. Therefore, overall correctness depends additionally on termination of the nested loops.

Foreach loop See `while` loop and B replaced with $(A \neq \emptyset)$.

Sequential composition Let WT_1 and WT_2 be workflow templates with labels $[l]\ WT_1; [k]\ WT_2\ [n]$, and $\varphi_1^{l,k}$ and $\varphi_2^{k,n}$ their respective logical representations. The premise is $\vdash_K^{CR} \{p\}\ WT_1;\ WT_2\, \{q\}$. For this premise to be true, we know by the proof rule for sequential composition that both $\vdash_K^{CR}\ \{p\}\ WT_1\, \{r\}$ and $\vdash_K^{CR}$ $\{r\}\ WT_2\, \{q\}$, and therefore such an expression $r \in \Phi_K$ has to exist. Let L^1 denote all loops of WT_1 and L^2 all loops of WT_2 (and L_{WT_1} and L_{WT_2} the corresponding top level loops according to definition). For a given loop $[h]$ *loop* B `do` $[i]\ W$ `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-PROG}_h \wedge \text{T-BOUND}_h$ for total correctness. Then:

$$
\begin{aligned}
& \vdash_K^{CR} \{p\}\ WT_1\, \{r\} \text{ and } \vdash_K^{CR} \{r\}\ WT_2\, \{q\} \\
\Rightarrow\ & \text{by assumption and induction base} \\
& \models_K^{CR} \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow r_k \wedge \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \\
& \quad \Big(m_k \wedge r_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big) \wedge \bigwedge_{loop_h \in L^2} VC_h \\
\Leftrightarrow\ & \text{by } \big(A \Rightarrow B \wedge C\big) \Leftrightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big) \\
& \models_K^{CR} \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow r_k\Big) \wedge \\
& \quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \wedge \\
& \quad \Big(m_k \wedge r_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big) \\
& \quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h \\
\Leftrightarrow\ & \text{by absorption } (A \Rightarrow B) \Leftrightarrow (A \Rightarrow B \wedge A) \\
& \models_K^{CR} \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow r_k \wedge m_l\Big) \wedge \\
& \quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \wedge \\
& \quad \Big(m_k \wedge r_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big) \\
& \quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}
$$

$\Rightarrow$ especially for $m_l = m_k$

$$\begin{aligned}
&\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow r_k \wedge m_l\Big) \wedge \\
&\quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \wedge \\
&\quad \Big(m_k \wedge r_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big) \\
&\quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}$$

$\Leftrightarrow$ by exportation $\big(A \wedge B \Rightarrow C\big) \Leftrightarrow \big(A \Rightarrow (B \Rightarrow C)\big)$

$$\begin{aligned}
&\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow r_k \wedge m_l\Big) \wedge \\
&\quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \wedge \\
&\quad \Big(m_k \wedge r_k \Rightarrow \big(\varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big) \\
&\quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}$$

$\Leftrightarrow$ by transitivity of implication and $m_l = m_k$

$$\begin{aligned}
&\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \big(\varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big) \wedge \\
&\quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \\
&\quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}$$

$\Leftrightarrow$ by exportation $\big(A \wedge B \Rightarrow C\big) \Leftrightarrow \big(A \Rightarrow (B \Rightarrow C)\big)$

$$\begin{aligned}
&\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big) \wedge \\
&\quad \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \Rightarrow \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \\
&\quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}$$

$$\begin{aligned}
\Leftrightarrow\ &\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \\
&\quad \bigwedge_{loop_i \in L_{WT_1}} (m_i \Rightarrow \mathbf{inv}_i)\Big) \\
&\quad \wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h
\end{aligned}$$

$$\Leftrightarrow \quad \models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2} \cup L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big)$$

$$\wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h$$

$\Leftrightarrow$ by Def. 5.6 (Loops, Invariants, and Termination Functions)

$$\models_K^{CR} \big(m_l = m_k\big) \wedge \Big(m_l \wedge p_l \wedge \varphi_1^{l,k} \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1;WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big)$$

$$\wedge \bigwedge_{loop_h \in L^1} VC_h \wedge \bigwedge_{loop_h \in L^2} VC_h$$

$$\Rightarrow \quad \models_K^{CR} \Big((m_l = m_k) \wedge m_l \wedge p_l \wedge \varphi_1^{l,k} \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1;WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big)$$

$$\wedge \bigwedge_{loop_h \in L^1 \cup L^2} VC_h$$

$\Leftrightarrow$ by Def. 5.5 (Logical Encoding of Workflow Templates)

$$\models_K^{CR} \Big(m_l \wedge p_l \wedge \varphi_{WT_1;WT_2}^{l,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1;WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big)$$

$$\wedge \bigwedge_{loop_h \in L^1 \cup L^2} VC_h$$

Conditional Let WT_1 and WT_2 be workflow templates, $\varphi_1^{i,n}$ and $\varphi_2^{k,n}$ their respective logical representations (with $\mathit{free}(\varphi_1^{i,n}) \subseteq \mathit{Var}_i \cup \mathit{Var}_n$ and $\mathit{free}(\varphi_2^{k,n}) \subseteq \mathit{Var}_k \cup \mathit{Var}_n$), and B a logical expression. Let l, i, k, n be labels as follows:
$[l]$ `if` B `then` $[i]$ WT_1 `else` $[k]$ WT_2 `fi` $[n]$.

The premise is $\vdash_K^{CR} \{p\}\, \texttt{ite}(B,\, WT_1,\, WT_2)\, \{q\}$. For the premise to be true, we know by the proof rule that $\vdash_K^{CR} \{p \wedge B\}\; WT_1\, \{q\}$ and $\vdash_K^{CR} \{p \wedge \neg B\}\; WT_2\, \{q\}$. Let L^1 denote all loops of WT_1 and L^2 all loops of WT_2 (and L_{WT_1} and L_{WT_2} the corresponding top level loops according to definition). For a given loop $[h]$ *loop* B `do` $[i]$ W `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-P}\textsc{rog}_h \wedge \text{T-B}\textsc{ound}_h$ for total correctness.

$$\vdash_K^{CR} \{p \wedge B\}\; WT_1\, \{q\} \text{ and } \vdash_K^{CR} \{p \wedge \neg B\}\; WT_2\, \{q\}$$

$\Rightarrow$ by premise and induction base

$$\models_K^{CR} \big(m_i \wedge p_i \wedge B_i \wedge \varphi^{i,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \wedge \bigwedge_{loop_j \in L^1} VC_j \wedge$$
$$\big(m_k \wedge p_k \wedge \neg B_k \wedge \varphi^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \wedge \bigwedge_{loop_j \in L^2} VC_j$$

$\Rightarrow$ we introduce m_l, Var_l, such that by Def. 5.2 (Variable Replacement ...) :

$$\models_K^{CR} \big(m_l \wedge m_l = m_i \wedge p_l \wedge B_l \wedge \bigwedge_{x \in \mathit{Var}} x_l = x_i \wedge \varphi^{i,n}$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \big(m_l \wedge m_l = m_k \wedge p_l \wedge \neg B_l \wedge \bigwedge_{x \in \mathit{Var}} x_l = x_k \wedge \varphi^{k,n}$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^1} VC_j \wedge \bigwedge_{loop_j \in L^2} VC_j$$

$\Rightarrow$ by $(A \Rightarrow B) \Rightarrow (A \wedge C \Rightarrow B)$:

$$\models_K^{CR} \big(m_l \wedge m_l = m_i \wedge \neg m_k \wedge p_l \wedge B_l \wedge \bigwedge_{x \in Var} x_l = x_i \wedge \varphi^{i,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \big(m_l \wedge m_l = m_k \wedge \neg m_i \wedge p_l \wedge \neg B_l \wedge \bigwedge_{x \in Var} x_l = x_k \wedge \varphi^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^1} VC_j \wedge \bigwedge_{loop_j \in L^2} VC_j$$

$\Rightarrow$ by Def. 5.6 (Loops), we know $\neg m_k \Rightarrow \bigwedge_{loop_j \in L_{WT_2}} \neg m_j$ (same for m_i):

$$\models_K^{CR} \big(m_l \wedge m_l = m_i \wedge \neg m_k \wedge p_l \wedge B_l \wedge \bigwedge_{x \in Var} x_l = x_i \wedge \varphi^{i,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \big(m_l \wedge m_l = m_k \wedge \neg m_i \wedge p_l \wedge \neg B_l \wedge \bigwedge_{x \in Var} x_l = x_k \wedge \varphi^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^1} VC_j \wedge \bigwedge_{loop_j \in L^2} VC_j$$

$\Leftrightarrow$ by Def. of implication and distribution

$$\models_K^{CR} \Big(\big(m_l \wedge m_l = m_i \wedge \neg m_k \wedge p_l \wedge B_l \wedge \bigwedge_{x \in Var} x_l = x_i \wedge \varphi^{i,n}\big)$$
$$\vee \big(m_l \wedge m_l = m_k \wedge \neg m_i \wedge p_l \wedge \neg B_l \wedge \bigwedge_{x \in Var} x_l = x_k \wedge \varphi^{k,n}\big)\Big)$$
$$\Rightarrow \big(q_n \wedge \bigwedge_{loop_j \in L_{WT_1 \cup WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^1 \cup L^2} VC_j$$

$\Leftrightarrow$ pull $m_l \wedge p_l$ by distribution:

$$\models_K^{CR} m_l \wedge p_l \wedge \Big(\big(m_l = m_i \wedge \neg m_k \wedge B_l \wedge \bigwedge_{x \in Var} x_l = x_i \wedge \varphi^{i,n}\big)$$
$$\vee \big(m_l = m_k \wedge \neg m_i \wedge \neg B_l \wedge \bigwedge_{x \in Var} x_l = x_k \wedge \varphi^{k,n}\big)\Big)$$
$$\Rightarrow \big(q_n \wedge \bigwedge_{loop_j \in L_{WT_1 \cup WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^1 \cup L^2} VC_j$$

$$\Leftrightarrow \quad \text{by Def. 5.5 (Logical Encoding of Workflow Templates)}$$
$$\models_K^{CR} m_l \land p_l \land \varphi^{l,n}_{\texttt{ite}(B, WT_1, WT_2)} \Rightarrow q_n \land \bigwedge_{loop_j \in L_{\texttt{ite}\ldots}} (m_j \Rightarrow \mathbf{inv}_j)$$
$$\land \bigwedge_{loop_j \in L^1 \cup L^2} VC_j$$

This proves the forward direction. □

Now, we prove the other direction of Theorem 5.10 (Provability corresponds with Tautology). We show that given a tautological formula $\models_K^{CR}$ Req based on a service composition template *Sct* with a workflow encoding φ_{WT} implies a provably correct workflow $\vdash_K^{CR} \{p\}\ WT\ \{q\}$ using labels $[l]\ WT\ [n]$.

We make use of the weakest (and weakest liberal) precondition of a workflow. We know that $\vdash_K^{CR} \{wlp(WT, q)\}\ WT\ \{q\}$, and the same for *wp* (Apt et al., 2009, and proof of Theorem 3.17). Therefore, it is sufficient to show that for a tautology $\models_K^{CR}$ Req with verification conditions *VC* of the structure $m_l \land p_l \land \varphi^{l,n}_{WT} \Rightarrow q_n \land \bigwedge_{loop_j \in L_{WT}} (m_j \Rightarrow \mathbf{inv}_j)$, the formula $p_l \Rightarrow wlp(WT, q)_l$ is also a tautology. We use the weakest (liberal) preconditions as defined in Lemma 3.19. For the proofs of the sequence and conditional statements we need the following lemma.

Lemma 5.11 (Variable Independence)**.** *Let $p, q \in \Phi_K$ over a set of variables Var. If $\models_K^{CR} p_i \land p_k \Rightarrow q_i$, then also $\models_K^{CR} p_i \Rightarrow q_i$.*

Proof. Be prove the lemma by contradiction. Assume that the lemma does not hold, that is, if $\models_K^{CR} p_i \land p_k \Rightarrow q_i$, then not necessarily $\models_K^{CR} p_i \Rightarrow q_i$. If $p_i \Rightarrow q_i$ is not valid, then there has to be a valuation such that $p_i \land \neg q_i$ is true (and $p_i \Rightarrow q_i$ is false). With this valuation, the only way for $p_i \land p_k \Rightarrow q_i$ to be valid is that p_k is false for every possible valuation. As $Var_i \cap Var_k = \emptyset$, there are no common variables between p_i and p_k and their valuations are independent. Therefore, if p_k is not contradictory, then there exists a valuation such that p_k is true. As $p \in \Phi_K$, p_k is not necessarily contradictory. Therefore, $\not\models_K^{CR} p_i \land p_k \Rightarrow q_i$, and the lemma holds. □

Now, we continue with the remaining proof of Theorem 5.10 (Provability corresponds with Tautology).

Proof. Let $[l]\ WT\ [n]$ be a workflow template and $\varphi^{l,n}_{WT}$ its logical encoding. We show that whenever Req is a tautology, $p \Rightarrow wlp(WT, q)$ (for partial correctness) and $p \Rightarrow wp(WT, q)$ (for total correctness) are also. We show this by induction on the structure of WT, using skip, assignment, service call, and loops as base cases, as the encoding of the latter is not defined inductively. For the base cases (except loops), L_{WT} is empty, so $\bigwedge_{loop_j \in L_{WT}} (m_j \Rightarrow \mathbf{inv}_j)$ is always true and we omit this part of the formula for easier reading.

Skip

$$
\begin{array}{ll}
 & \models^{CR}_{K} m_l \wedge p_l \wedge \varphi^{l,n}_{skip} \Rightarrow q_n \\
\Rightarrow & \text{we are only interested in active workflows (Def. 5.4, Position Markers)} \\
 & \models^{CR}_{K} \big(m_l \wedge p_l \wedge \varphi^{l,n}_{skip} \Rightarrow q_n\big) \wedge m_l \\
\Leftrightarrow & \models^{CR}_{K} p_l \wedge \varphi^{l,n}_{skip} \Rightarrow q_n \\
\Leftrightarrow & \text{by Def. 5.5 (Logical Encoding of Workflow Templates)} \\
 & \models^{CR}_{K} p_l \wedge \bigwedge_{x \in Var} x_l = x_n \Rightarrow q_n \\
\Leftrightarrow & \models^{CR}_{K} p_l \Rightarrow q_l \\
\Leftrightarrow & \text{by Lemma 4.18 (wlp of Skip)} \\
 & \models^{CR}_{K} p_l \Rightarrow wlp(skip, q)_l
\end{array}
$$

The same is true for $wp(skip, q)$.

Assignment

$$
\begin{array}{ll}
 & \models^{CR}_{K} m_l \wedge p_l \wedge \varphi^{l,n}_{u:=t} \Rightarrow q_n \\
\Rightarrow & \text{we are only interested in active workflows (Def. 5.4, Position Markers)} \\
 & \models^{CR}_{K} \big(m_l \wedge p_l \wedge \varphi^{l,n}_{u:=t} \Rightarrow q_n\big) \wedge m_l \\
\Leftrightarrow & \models^{CR}_{K} p_l \wedge \varphi^{l,n}_{u:=t} \Rightarrow q_n \\
\Leftrightarrow & \text{by Def. 5.5 (Logical Encoding of Workflow Templates)} \\
 & \models^{CR}_{K} \big(p_l \wedge u_n = t_l \wedge \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n\big) \Rightarrow q_n \\
\Leftrightarrow & \text{by Def. of Substitution and } \mathcal{I}(u_n) = \mathcal{I}(t_l) \\
 & \models^{CR}_{K} p_l \Rightarrow q[u := t]_l \\
\Leftrightarrow & \text{by Lemma 4.18 (wlp of Assignment)} \\
 & \models^{CR}_{K} p_l \Rightarrow wlp(u := t, q)_l
\end{array}
$$

The same is true for $wp(u := t, q)$.

Service Call (partial correctness)

$$\models_K^{CR} m_l \land p_l \land \varphi_{u:=S(v)} \Rightarrow q_n$$

$\Rightarrow$ we are only interested in active workflows (Def. 5.4, Position Markers)

$$\models_K^{CR} \big(m_l \land p_l \land \varphi^{l,n}_{u:=S(v)} \Rightarrow q_n\big) \land m_l$$

$$\Leftrightarrow \quad \models_K^{CR} p_l \land \varphi^{l,n}_{u:=S(v)} \Rightarrow q_n$$

$\Leftrightarrow$ by Def. 5.5 (Service Call)

$$\models_K^{CR} \Big(p_l \land (post[i := v, o := u]_n \land \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n)\Big) \Rightarrow q_n$$

$\Leftrightarrow$ by exportation $\big(A \land B \Rightarrow C\big) \Leftrightarrow \big(A \Rightarrow (B \Rightarrow C)\big)$

$$\models_K^{CR} p_l \Rightarrow \Big((post[i := v, o := u]_n \land \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n) \Rightarrow q_n\Big)$$

$\Rightarrow$ because $u_n \notin free(p_l), u_n \notin free(pre[i := v]_l)$ by Def. 5.2

$$\models_K^{CR} p_l \Rightarrow$$
$$\Big(\big(\forall w : post[i := v, o := w]_n \land \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n\big) \Rightarrow q[u := w]_n\Big)$$

$\Rightarrow$ because u is no longer part of any formula

$$\models_K^{CR} p_l \Rightarrow \Big(\forall w : post[i := v, o := w]_l \Rightarrow q[u := w]_l\Big)$$

$\Leftrightarrow$ by Lemma 4.18

$$\models_K^{CR} p_l \Rightarrow wlp(u := S(v), q)_l$$

Service Call (total correctness)

$$\models_K^{CR} m_l \wedge p_l \wedge \varphi_{u:=S(v)} \Rightarrow q_n$$

$\Rightarrow$ we are only interested in active workflows (Def. 5.4, Position Markers)

$$\models_K^{CR} \left(m_l \wedge p_l \wedge \varphi^{l,n}_{u:=S(v)} \Rightarrow q_n\right) \wedge m_l$$

$\Leftrightarrow$ $\models_K^{CR} p_l \wedge \varphi^{l,n}_{u:=S(v)} \Rightarrow q_n$

$\Leftrightarrow$ by Def. 5.5 (Service Call)

$$\models_K^{CR} \Big(p_l \wedge \big(pre[i := v]_l \Rightarrow (post[i := v, o := u]_n \wedge \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n)\big)\Big) \Rightarrow q_n$$

$\Leftrightarrow$ by exportation $\big(A \wedge B \Rightarrow C\big) \Leftrightarrow \big(A \Rightarrow (B \Rightarrow C)\big)$

$$\models_K^{CR} p_l \Rightarrow \Big(\big(pre[i := v]_l \Rightarrow (post[i := v, o := u]_n \wedge \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n)\big) \Rightarrow q_n\Big)$$

$\Rightarrow$ by $\big((A \Rightarrow B) \Rightarrow C\big) \Rightarrow \big(A \wedge B \Rightarrow C\big)$

$$\models_K^{CR} p_l \Rightarrow \big((pre[i := v]_l \wedge post[i := v, o := u]_n \wedge \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n) \Rightarrow q_n\big)$$

$\Rightarrow$ because $u_n \notin free(p_l), u_n \notin free(pre[i := v]_l)$ by Def. 5.2

$$\models_K^{CR} p_l \Rightarrow$$
$$\Big(\big(pre[i := v]_l \wedge \forall w : post[i := v, o := w]_n \wedge \bigwedge_{x \in Var \setminus \{u\}} x_l = x_n\big) \Rightarrow q[u := w]_n\Big)$$

$\Rightarrow$ because u is no longer part of any formula

$$\models_K^{CR} p_l \Rightarrow \Big(pre[i := v]_l \wedge \forall w : post[i := v, o := w]_l \Rightarrow q[u := w]_l\Big)$$

$\Leftrightarrow$ by Lemma 4.18

$$\models_K^{CR} p_l \Rightarrow wp(u := S(v), q)_l$$

While loop Let WT be a workflow and B a loop condition. We choose $l \neq k \neq n$ as labels as in $[l]$ `while` B `do` $[k]$ WT `od` $[n]$. Also, let `while` be a short-hand for `while` B `do` WT `od`, and $X := \mathit{Var} \setminus \mathit{change}(WT)$, and W the body of inner loops. For a given loop $[h]$ *loop* B `do` $[i]$ W `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-P}\textsc{rog}_h \wedge \text{T-B}\textsc{ound}_h$ for total correctness.

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT}} (m_j \Rightarrow \mathbf{inv}_j)\big) \wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ because the loop is the single element of L_{WT} (Def. 5.6)

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow q_n \wedge (m_l \Rightarrow \mathbf{inv}_l)\big) \wedge \bigwedge_{loop_j \in L} VC_j$$

$\Rightarrow$ we are only interested in active workflows (Def. 5.4, Position Markers)

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow q_n \wedge (m_l \Rightarrow \mathbf{inv}_l)\big) \wedge m_l \wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ $\models_K^{CR} \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow q_n \wedge \mathbf{inv}_l\big) \wedge \bigwedge_{loop_j \in L} VC_j$

$\Leftrightarrow$ by $\big(A \Rightarrow B \wedge C\big) \Leftrightarrow \big((A \Rightarrow B) \wedge (A \Rightarrow C)\big)$

$$\models_K^{CR} \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow q_n\big) \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ by Def. 5.5 (While)

$$\models_K^{CR} \big(p_l \wedge \mathbf{inv}_l \wedge \mathbf{inv}_n \wedge \neg B_n \wedge \bigwedge_{x \in X} (x_l = x_n) \Rightarrow q_n\big)$$
$$\wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \wedge \bigwedge_{loop_j \in L} VC_j$$

$\Rightarrow$ from $\vdash \{\mathbf{inv}\}\,\texttt{while}\ldots\{\mathbf{inv} \wedge \neg B\}, \vdash \{p\}\,W\,\{q\}$ and $\models p \Rightarrow wp(W, q)$:

$$\models_K^{CR} \big(p_l \wedge \mathbf{inv}_l \wedge \mathbf{inv}_n \wedge \neg B_n \wedge \bigwedge_{x \in X} (x_l = x_n) \Rightarrow q_n\big)$$
$$\wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big)$$
$$\wedge \big(\mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, \mathbf{inv}_n \wedge \neg B_n)\big) \wedge \bigwedge_{loop_j \in L} VC_j$$

$$
\begin{aligned}
\Leftrightarrow\quad & \text{by exportation } \big(A \wedge B \Rightarrow C\big) \Leftrightarrow \big(A \Rightarrow (B \Rightarrow C)\big) \\
& \models_K^{CR} \big(p_l \wedge \mathbf{inv}_l \wedge \bigwedge_{x\in X} (x_l = x_n) \Rightarrow (\mathbf{inv}_n \wedge \neg B_n \Rightarrow q_n)\big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \big(\mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, \mathbf{inv}_n \wedge \neg B_n)\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Rightarrow\quad & \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \wedge \bigwedge_{x\in X} (x_l = x_n) \Rightarrow \\
& \quad \big((\mathbf{inv}_n \wedge \neg B_n \Rightarrow q_n) \wedge wp(\texttt{while}\ldots, \mathbf{inv}_n \wedge \neg B_n)\big)\Big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Rightarrow\quad & \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \wedge \bigwedge_{x\in X} (x_l = x_n) \Rightarrow wp(\texttt{while}\ldots, q_n)\Big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Rightarrow\quad & \text{especially true for } l = n \\
& \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, q_l)\Big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j
\end{aligned}
$$

As we can see, p does not always imply the weakest precondition, as we need to prove. The proof depends on whether or not we chose a "useful" invariant to encode the loop statement. However, concerning $p \Rightarrow wp(\ldots, q)$, we are only interested in the cases where the variable assignment represents an actual workflow. More precisely, if $\varphi_{\texttt{while}}$ is false, it is irrelevant whether p implies the weakest precondition. In contrast, if $\varphi_{\texttt{while}}$ is true, p has to imply wp. Therefore:

$$
\begin{aligned}
& \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, q_l)\Big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Rightarrow \quad & \text{we are only interested in the case } \varphi^{l,n}_{\texttt{while}} = \top \\
& \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, q_l)\Big) \\
& \quad \wedge \big(p_l \wedge \varphi^{l,n}_{\texttt{while}} \Rightarrow \mathbf{inv}_l\big) \wedge \varphi^{l,n}_{\texttt{while}} \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Leftrightarrow \quad & \models_K^{CR} \Big(p_l \wedge \mathbf{inv}_l \Rightarrow wp(\texttt{while}\ldots, q_l)\Big) \\
& \quad \wedge \big(p_l \Rightarrow \mathbf{inv}_l\big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j \\
\Leftrightarrow \quad & \models_K^{CR} \Big(p_l \Rightarrow wp(\texttt{while}\ldots, q_l)\Big) \\
& \quad \wedge \bigwedge_{loop_j \in L} VC_j
\end{aligned}
$$

Foreach loop See `while` loop, with B replaced by $A \neq \emptyset$.

Sequential composition Let WT_1 and WT_2 be workflow templates, and $\varphi_1^{l,k}$ and $\varphi_2^{k,n}$ their respective logical representations. We use $l \neq k \neq n$ as labels as in $[l]\ WT_1; [k]\ WT_2\ [n]$. Let L^1 denote all loops of WT_1 and L^2 all loops of WT_2 (and L_{WT_1} and L_{WT_2} the corresponding top level loops according to definition). For a given loop $[h]$ *loop* B `do` $[i]$ W `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-P}\textsc{rog}_h \wedge \text{T-B}\textsc{ound}_h$ for total correctness.

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1;WT_2}^{l,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1;WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^{WT_1;WT_2}} VC_j$$

$\Leftrightarrow$ by Def. 5.5

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1;WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^{WT_1;WT_2}} VC_j$$

$\Leftrightarrow$ by Def. 5.6 (Loops, Invariants, and Termination Functions), two times

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n}$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L^{WT_1}} VC_j \bigwedge_{loop_j \in L^{WT_2}} VC_j$$

$\Leftrightarrow$ by $(A \Rightarrow B \wedge C) \Leftrightarrow (A \Rightarrow B) \wedge (A \Rightarrow C)$

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n}$$
$$\Rightarrow \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n} \qquad (1)$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \qquad (1)$$
$$\wedge \bigwedge_{loop_j \in L^{WT_1}} VC_j \bigwedge_{loop_j \in L^{WT_2}} VC_j$$

Now, for conjunct (1), we apply Craig interpolation.

$$
\begin{aligned}
& \models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n} \\
& \qquad \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
\Leftrightarrow \quad & \text{by } m_l \wedge m_l = m_k \text{ and } (\models_K^{CR} p) \Leftrightarrow (\neg p \text{ UNSAT}) \\
& \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_k \wedge \varphi_{WT_2}^{k,n} \\
& \qquad \wedge \neg\big(q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\big) \qquad \text{UNSAT} \\
\Rightarrow \quad & \text{by Craig interpolation } \exists r_k \text{ such that:} \\
& \Big(\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \Rightarrow r_k\big)\Big) \\
& \text{and } \Big(r_k \wedge m_k \wedge \varphi_{WT_2}^{k,n} \wedge \neg\big(q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big) \qquad \text{UNSAT} \\
\Leftrightarrow \quad & \text{by } (\models_K^{CR} p) \Leftrightarrow (\neg p \text{ UNSAT}) \\
& \models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \Rightarrow r_k\big) \\
& \quad \wedge \Big(r_k \wedge m_k \wedge \varphi_{WT_2}^{k,n} \Rightarrow \big(q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big)
\end{aligned}
$$

We combine this result for (1) with the overall proof:

$$
\begin{aligned}
\Rightarrow \quad & \text{by Craig interpolation} \\
& \models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \wedge m_l = m_k \wedge \varphi_{WT_2}^{k,n} && (2) \\
& \qquad \Rightarrow \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) && (2) \\
& \quad \wedge \big(m_l \wedge p_l \wedge \varphi_{WT_1}^{l,k} \Rightarrow r_k\big) && (1) \\
& \quad \wedge \Big(r_k \wedge m_k \wedge \varphi_{WT_2}^{k,n} \Rightarrow \big(q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big) && (1) \\
& \quad \wedge \bigwedge_{loop_j \in L^{WT_1}} VC_j \bigwedge_{loop_j \in L^{WT_2}} VC_j
\end{aligned}
$$

For conjunct (2), from the structure and labeling of the sequential workflow we know that the consequent does not include any variable $x_k \in Var_k$. Therefore we can apply Lemma 5.11 (Variable Independence) and simplify the antecedent.

$\Rightarrow$ by Lemma 5.11 (Variable Independence)

$$\begin{aligned}
&\models_K^{CR} \big(m_l \land p_l \land \varphi_{WT_1}^{l,k} \Rightarrow \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \qquad (2)\\
&\quad \land \big(m_l \land p_l \land \varphi_{WT_1}^{l,k} \Rightarrow r_k\big)\\
&\quad \land \Big(r_k \land m_k \land \varphi_{WT_2}^{k,n} \Rightarrow \big(q_n \land \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big)\\
&\quad \land \bigwedge_{loop_j \in L^{WT_1}} VC_j \bigwedge_{loop_j \in L^{WT_2}} VC_j
\end{aligned}$$

$\Leftrightarrow$ by $(A \Rightarrow B) \land (A \Rightarrow C) \Leftrightarrow (A \Rightarrow B \land C)$

$$\begin{aligned}
&\models_K^{CR} \big(m_l \land p_l \land \varphi_{WT_1}^{l,k} \Rightarrow r_k \land \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big)\\
&\quad \land \Big(r_k \land m_k \land \varphi_{WT_2}^{k,n} \Rightarrow \big(q_n \land \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)\Big)\\
&\quad \land \bigwedge_{loop_j \in L^{WT_1}} VC_j \bigwedge_{loop_j \in L^{WT_2}} VC_j
\end{aligned}$$

$\Leftrightarrow$ by induction hypothesis

$\{p\}\, WT_1 \,\{r\}, \{r\}\, WT_2 \,\{q\}$

$\Rightarrow$ by rule of composition

$\{p\}\, WT_1; WT_2 \,\{q\}$

Conditional Let WT_1, WT_2 be workflow templates, $\varphi_1^{h,n}$, $\varphi_2^{k,n}$ their respective logical representations, and B the logical condition. We choose $l \neq h \neq k \neq n$ as labels as in
$[l]$ `if` B `then` $[h]$ WT_1 `else` $[k]$ WT_2 `fi` $[n]$. Also, let `if` be a short-hand for
`if` B `then` WT_1 `else` WT_2 `fi`, and W the body of an inner loop. For a given loop $[h]$ *loop* B `do` $[i]$ W `od` $[j]$ we write VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv})$ for partial correctness, and VC_h instead of $VC(W, \mathbf{inv} \wedge B, \mathbf{inv}) \wedge \text{T-P}\textsc{rog}_h \wedge \text{T-B}\textsc{ound}_h$ for total correctness.

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{\texttt{if}}^{l,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ by Def. 5.6

$$\models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{\texttt{if}}^{l,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1 \cup WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L} VC_j$$

$$\Leftrightarrow \quad \models_K^{CR} \big(m_l \wedge p_l \wedge \varphi_{\texttt{if}}^{l,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ by Def. 5.5 (Conditional)

$$\models_K^{CR} \Big(\big((m_l \wedge p_l \wedge m_l = m_h \wedge \neg m_k \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_1^{h,n})$$
$$\vee (m_l \wedge p_l \wedge m_l = m_k \wedge \neg m_h \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_2^{k,n})\big)$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\Big)$$
$$\wedge \bigwedge_{loop_j \in L} VC_j$$

$\Leftrightarrow$ by $(A \vee B \Rightarrow C) \Leftrightarrow (A \Rightarrow C) \wedge (B \Rightarrow C)$

$$\models_K^{CR} \big(m_l \wedge p_l \wedge m_l = m_h \wedge \neg m_k \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_1^{h,n}$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \big(m_l \wedge p_l \wedge m_l = m_k \wedge \neg m_h \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_2^{k,n}$$
$$\Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j) \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big)$$
$$\wedge \bigwedge_{loop_j \in L} VC_j$$

$\Rightarrow$ by $(A \Rightarrow B \wedge C) \Rightarrow (A \Rightarrow B)$

$$\begin{aligned}
&\models^{CR}_{K} \big(m_l \wedge p_l \wedge m_l = m_h \wedge \neg m_k \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_1^{h,n} \\
&\qquad \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \big(m_l \wedge p_l \wedge m_l = m_k \wedge \neg m_h \wedge B_l \wedge \bigwedge_{x \in Var} (x_l = x_h) \wedge \varphi_2^{k,n} \\
&\qquad \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \bigwedge_{loop_j \in L} VC_j
\end{aligned}$$

$$\begin{aligned}
\Leftrightarrow \quad &\models^{CR}_{K} \big(m_h \wedge p_h \wedge \neg m_k \wedge B_h \wedge \varphi_1^{h,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \big(m_k \wedge p_k \wedge \neg m_h \wedge B_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \bigwedge_{loop_j \in L} VC_j
\end{aligned}$$

$\Rightarrow$ by Lemma 5.11 (Variable Independence) we can omit $\neg m_k, \neg m_h$

$$\begin{aligned}
&\models^{CR}_{K} \big(m_h \wedge p_h \wedge B_h \wedge \varphi_1^{h,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \big(m_k \wedge p_k \wedge B_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \bigwedge_{loop_j \in L} VC_j
\end{aligned}$$

$\Leftrightarrow$ by Def 5.6 (Loops, Invariants, and Termination Functions)

$$\begin{aligned}
&\models^{CR}_{K} \big(m_h \wedge p_h \wedge B_h \wedge \varphi_1^{h,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \big(m_k \wedge p_k \wedge B_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \\
&\quad \wedge \bigwedge_{loop_j \in L^1} VC_j \wedge \bigwedge_{loop_j \in L^2} VC_j
\end{aligned}$$

$$\Leftrightarrow \quad \models_K^{CR} \big(m_h \wedge p_h \wedge B_h \wedge \varphi_1^{h,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_1}} (m_j \Rightarrow \mathbf{inv}_j)\big) \wedge \bigwedge_{loop_j \in L^1} VC_j \ ,$$

$$\models_K^{CR} \big(m_k \wedge p_k \wedge B_k \wedge \varphi_2^{k,n} \Rightarrow q_n \wedge \bigwedge_{loop_j \in L_{WT_2}} (m_j \Rightarrow \mathbf{inv}_j)\big) \wedge \bigwedge_{loop_j \in L^2} VC_j$$

$\Rightarrow$ by induction

$\vdash_K^{CR} \{p \wedge B\}\ WT_1\,\{q\}, \vdash_K^{CR} \{p \wedge \neg B\}\ WT_2\,\{q\}$

$\Rightarrow$ by conditional rule

$\{p\}$ `if` B `then` WT_1 `else` WT_2 `fi` $\{q\}$

□

This concludes the proof of Theorem 5.10 (Provability corresponds with Tautology) and therefore establishes a connection between solving a satisfiability problem and the constructibility of a formal proof of correctness.

5.4 Deriving Invariants and Termination Functions

Automatically discovering loop invariants and termination functions is a well researched topic and not part of this thesis. However, as the integration of formalized domain knowledge in the process of verification of service compositions is the core part of this thesis, we present first strategies to utilize a knowledge base in the process of finding invariants and termination functions.

5.4.1 Finding Loop Invariants using Domain Knowledge

Section 5.1 provides an overview of invariant generation techniques, with two main topics: Invariants formalize either linear inequalities between (numerical) variables and a constant, or they are derived from an existing postcondition of the loop by mutation, utilizing heuristics. A third approach is the use of invariant templates, where a pattern of the invariant formula is already given and the use of actual predicates is computed dynamically. As an example, Srivastava and Gulwani provide three example patterns of invariants:

$$\begin{aligned} &\nu_1 \\ \forall y : \; &\nu_1 \Rightarrow \nu_2 \\ \forall y \exists x : \; &\nu_1 \Rightarrow \nu_2 \end{aligned}$$

Here, ν is a variable representing a set (treated as conjunction) of predicates, which is determined by actual invariant inference algorithms (Srivastava and Gulwani, 2009). However, the predicates are given manually by the user and take the form of linear inequalities ($x < 5$, $y < n$) or array access ($A[j] = 0$), that is, these are not predicates in the sense of a knowledge base in this thesis. Due to the nature of the predicates, this approach does not seem to be directly applicable to the invariants needed in this thesis. However, the actual computation of ν depends on fix point computations which rely on an SMT solver. Therefore, this approach is not *per se* limited in the form of the predicates as long as there exists a representation which can be handled by the underlying solver.

There are two main differences between the predicates handled by the approach of Srivastava and Gulwani and this thesis: One is the use of real first-order logic predicates, the other is the use of a type system for variables, including set types. Therefore, defining ν as $\nu \in 2^{\mathcal{P}}$, the first difference is already balanced. To compensate for the second difference, namely the use of set types, we can use invariant templates. Invariant templates are not necessarily as abstract and simply structured as in the example above. Instead, they incorporate structural knowledge of the loop designer about an invariant, even if the exact invariant is not known.

Considering the Filter template ($\rightarrow$ Example 4.4), where the output is a subset of the input for which a certain predicate holds, we know the following about the loop:

- There is an input set A, an output set B, and a temporary set Z, which serves as the loop variable.
- The temporary set is always a subset of the input set.
- The output set contains all elements of the input set which have already been processed by the loop, as long as the target predicate holds.
- From the precondition of the template we also know that an additional predicate holds for every element of the input set.

This knowledge leads to the invariant which is used in the manual proof of correctness (→ Figure 4.3):

$$Z \subseteq A \wedge \big(\forall u \in Z : \mathit{pre}_{\mathit{Acquire}}(u)\big) \wedge B = \{b \in A \setminus Z \mid \mathit{isTargetElement}(b)\}$$

While this invariant is specific to the loop of the Filter template, its structure follows a pattern outlined textually above:

$Z \subseteq A$	Relate loop set with a set variable,
$\forall u \in Z : p(u)$	assert a property $p \in \mathcal{P}$ for all set elements,
$B = \{b \in A \setminus Z \mid q(b)\}$	relate the output set with the input set by a predicate q.

Patterns with this or a similar structure rely on the set of (first-order) predicates as well as set operations. Template-based invariant inference as in the approach of Srivastava and Gulwani, which is inherently agnostic of the exact type of predicates used, are then applicable to find invariants based on the knowledge base and terms defined in this thesis. This is due to the fact that the underlying solver handles both the theory of uninterpreted functions, quantification, and there is an encoding of the handling of set types.

5.4.2 Finding Termination Functions using Domain Knowledge

Whenever verification of a workflow with respect to its postcondition includes verification of a loop, proving *termination* is an important issue. Termination of loops is typically shown by giving a *termination argument*, also called *termination function*, or, in this thesis, *termination expression*. A basic idea of proving termination of a program dates back to Turing. He suggests to represent the progress between program states as decreasing and eventually “vanishing” ordinal numbers (Morris and Jones, 1984). The mathematical idea behind this is to define *ranking functions* to map relations between program states to relations on a well ordered set, where a well ordered set is a set with a total order and a bound (Bradley et al., 2005, Chawdhary et al., 2008, Dams et al., 2000, Podelski and Rybalchenko, 2004). This is often done by using the non-negative Integers.

Using the notation of the logical encoding introduced in this chapter, a termination argument $\mathbf{t}$ is modified by each loop iteration such that it *decreases*, that is, if $\mathbf{t} = z$

for some z before the execution of the loop body, then $\mathtt{t} < z$ holds after executing the loop body. Additionally, $\mathtt{t}$ is *bounded* ($\mathtt{t} \geq k$ for some k) such that it cannot decrease (or progress) indefinitely. In this thesis, termination arguments are defined manually, if necessary, and are expressions with $\mathtt{t} \in \Phi_K$ and $\mathit{type}(\mathtt{t}) = T$. While T is often represented as Integer, it can be any range with an equality relation $=$ and a relation $<$ that induces a strict order on T. Note that the bound k is not necessarily a greatest lower bound in terms of a lattice, but can also be derived either from the type (e.g., zero for non-negative Integers), the domain knowledge, or the statements in the loop body.

Both, progress and boundedness, are part of the proof rule for `while` loops in case of the proof system for total correctness ($\rightarrow$ Definition 4.12, Parameterized Proof Calculus for Workflow Templates, rule 7.2). The `foreach` loop does not need a termination function, as its termination is implicitly included in its definition. This follows from the semantics of the *take* statement and the restriction of loop variables: For a loop `foreach` $a \in A$ `do` W `od`, the set A cannot be modified except by the semantics of the loop and, consequently, the *take* statement. Then, the termination argument $\mathtt{t}$ is the size of A and therefore progressing (by the semantics of *take*) towards a bound (the empty set).

The integration of domain knowledge follows the same principles. At first, termination argument candidates are found, at second, they are either validated or invalidated as termination argument. Candidates are derived from predicates and functions of the domain knowledge K. Any type $T \in \mathcal{T}_K$ with a relation $F \in \mathcal{F}_K$ or predicate $P \in \mathcal{P}_K$ inducing a strict order is a potential type of $\mathtt{t}$. Using the Tourism domain as an example ($\rightarrow$ Example 2.1), the concept $\textsc{Rating}$ is accompanied by a role $\textsc{isRatingLess} : \textsc{Rating} \times \textsc{Rating}$, with properties that induce a strict order on $\textsc{Rating}$ (with the equality relation $=$). Then, $\textsc{isRatingLess}$ corresponds with the order-inducing relation $<$ used in $\mathtt{t} < z$. Consequently, $\mathit{type}(\mathtt{t}) = \textsc{Rating}$. Not only predicates from the domain knowledge, but relations from $\mathcal{P}_K$ and $\mathcal{F}_K$ are also eligible candidates for order-inducing relations, e.g., the subset relation for any set type.

Following this pattern, a number of candidates for termination expression types, corresponding relations, and (simple) expression candidates can be derived directly from the knowledge base. Then, composite expression candidates can be built based on combining applicable predicates, functions, and variables. Simple heuristics would involve the use of variables in the loop body, and building expressions involving them and the relations and functions applicable to their type(s).

As this heuristic aims at identifying a well-founded structure in the domain knowledge and derived types, it should be relatively easy to adapt existing approaches to find termination arguments, as long as they are also based on well-foundedness, as, e.g., Dams et al. (2000).

Chapter 6

Dealing with Uncertain Service Descriptions

In service-oriented computing, especially OTF computing, services used in a service composition originate typically from third-party developers. The orchestration process, and, consequently, formal verification of the resulting service composition model, rely on their black-box descriptions. This poses the problem of services whose behavior may deviate from their description. In this thesis, we have the general assumption that service descriptions always reflect their actual behavior. This results from either the service being a composition itself (and being verified against its description), or an atomic service or piece of software. The latter can be treated using classical program verification.

This chapter discusses an approach to deal with *uncertain*, or deceptive, service compositions. It is based on the master's thesis of Maryam Lexow (née Sanati) and an unpublished paper of its results (Sanati, 2014).

6.1 Uncertainty in Service Descriptions

On a service market, we assume that service descriptions accurately reflect the actual behavior of the service. This can be achieved by techniques like *proof carrying code* (Necula and Lee, 1998) or *programs from proofs* (Wonisch et al., 2013a, 2014), which both create a proof of correctness which can be easily verified and thus serves as a certificate. Then, verification as presented in this thesis, works on the *model* of the behavior of the service. However, if services are not certified, then a correspondence between their model and their actual behavior cannot be guaranteed. Consequently, using their description to verify service compositions may lead to incorrect results, as the descriptions of single services may be unsound and do not represent their actual behavior. This raises the question of whether we can extend our verification framework such that it accounts for unreliability of formal descriptions.

A way to handle uncertainty in the correspondence of service models and their actual behavior is the use of probabilities. It raises two main questions:

(1) How is uncertainty modeled (and, especially, what is the source of this model)?

(2) What is its impact on verification?

In the context of on-the-fly computing, statistics are a potential source of probabilities to quantify the model/behavior correspondence. Services and service compositions are run on compute centers, and non-functional metrics like mean runtime, mean number of users, or mean failure rate can be tracked. If we consider runtime statistics as a source to quantify model/behavior correspondence statistics, the main questions are:

(1) Are they relevant in comparison to the goal of verification?

(2) Are they available in the verification process?

(3) Are they created independently of the original service provider (and not biased by it)?

However, our verification framework is focused on *functional* models describing the behavior of a service. The *users* of a service (or composition) are another source which can testify whether or not the service met the initial requirements. Typically, users provide feedback utilizing feedback systems, which in the end accumulate a *reputation* for a service or a service provider. In general, *reputation systems* (Mármol and Kuhnen, 2015) provide a value (or a vector of values) representing the experiences of former uses of the services. They serve as a means of decision making to select services for a composition. While reputation values do not rely on measurable statistics comparable to, e.g., runtime values, they include a direct feedback of customer satisfaction.

Both approaches, using statistical runtime data and subjective satisfaction feedback, have one thing in common: They both try to quantify a sliding scale of model/behavior correspondence. However, if services of a composition are not rigorously certified, but selected on the basis of statistical (empirical or subjectiv) data, the results of formal verification cannot be taken at face value. The reason for this is that, e.g., non-optimal reputation may include the fact that the service's behavior deviated from its published description, depending on the exact type of reputation value.

While uncertain verification results in general are well understood in Markov-based service models (e.g., state-based model checking with Prism, Kwiatkowska and Parker, 2012), or in the domain of quality-of-service analysis (Gallotti et al., 2008), SAT-based analysis of service compositions still relies on correct service descriptions. However, while it is out of scope of this thesis to define a verification framework which includes statistics, we will present an idea of combining a statistical approach with SAT-based analysis.

6.2 Verification under Uncertainty

To combine uncertain (or unreliable) service descriptions with formal verification the following steps are necessary:

(1) Based on empirical data, determine a *probability of correctness* of a service description;

(2) determine a service representation if it is *not* correct, that is, it does not match the actual behavior of the service;

(3) use the probability of correctness as part of formal verification.

Step (1) is the most flexible step. Its core is a (reliable) relation between empirical data with a probability of whether or not a service description can be trusted. Defining such a relation is not straightforward and not part of this thesis. However, the choices made in this step have major implications on the interpretation of verification results. With no regard to the exact mapping, a *probability of correctness* (PC) maps empirical data (e.g., aggregated reputation values, as in Jungmann et al. 2014, or failure statistics) to a probability value:

$$PC\colon \text{empirical data} \rightarrow [0;1] \ .$$

It defines the probability that a service *description* actually represents the service *behavior*. The logical encoding φ_W of a service composition's workflow W is defined inductively ($\rightarrow$ Definition 5.5, Logical Encoding of Workflow Templates), representing the service call $[l]\ u := S(v)\ [k]$. To include a probability of correctness, such a fixed definition can be replaced by a variable to represent different service call encodings. For example, for a service call $[l]\, u := S(v)\, [k]$, the special variable φ_S° may replace the encoding of a service call of Definition 5.5:

$$\varphi_S^{l,k} := \varphi_S^{\circ,l,k} \ .$$

As a result, the logical representation φ_{Sc} of a service composition Sc contains only placeholder variables as service representations. With a logical representation of a composition using service placeholders, we are able to define a domain for the placeholder variables. As part of step (2), a minimal domain would contain the original service call encoding and an additional encoding to represent the service behavior if it is not true to its description. In

$$\mathcal{U}(\varphi_S^\circ) := \{\varphi_S^{\checkmark}, \varphi_S^{\lightning}\} \ ,$$

$\varphi_S^{\checkmark}$ represents the service call encoding which represents the correct behavior, and $\varphi_S^{\lightning}$ represents a deviating behavior. As an example, a deviating service behavior may be modeled as the correct service, but with negated postcondition. That way, we get different representations for φ_S°:

$$\begin{aligned}
\varphi_{u:=S(v)}^{\checkmark,l,k} &:= \quad pre[i := v]_l \Rightarrow \big(post[i := v, o := u]_k \wedge \bigwedge_{x \in Var \setminus \{u\}} x_k = x_l\big) \\
\varphi_{u:=S(v)}^{\lightning,l,k} &:= \quad pre[i := v]_l \Rightarrow \big(\neg post[i := v, o := u]_k \wedge \bigwedge_{x \in Var \setminus \{u\}} x_k = x_l\big)
\end{aligned}$$

Other logical representations for $\varphi_S^{\lightning}$ are also possible. Different logical encodings for φ_S° combined with the probability of correctness enable a probability distribution D_S for φ_S° valuations:

$$D_S := \big\{\big(\varphi_S^{\checkmark}, PC(S)\big), \big(\varphi_S^{\lightning}, 1 - PC(S)\big)\big\} \ .$$

Using variables to represent logical service encodings, an encoding φ_{WT} for a given workflow template cannot be verified directly. Instead, service call variables φ_S° have to be instantiated before an actual verification can be executed. The general approach to step (3) is to create logical workflow template encodings by instantiating all possible permutations of φ_S° valuations. Every single permutation is a complete workflow template encoding with service call representations and therefore it can be automatically verified as discussed in Chapter 5 (Automating Correctness Proofs using First-order Logic).

Based on the probability distribution D_S, the *probability of occurrence* of every permutation can be computed. Additionally, the verification of every permutation yields a verification result. The goal would be to aggregate both information to an overall probability of correctness of the workflow template. However, it is not possible to weigh the results (that is, “correct”, “not correct”, or “unknown”) with the probability of occurrence, because there are two different notions of probability involved: The probability of correctness *of service descriptions* quantifies the correctness of a single service representation. Based on that, we can compute the probability of occurrence of a specific *workflow template encoding* as a permutation of service call representations.

To compute an overall probability of correctness, Littman et al. with Stochastic SAT and, based on their work, Fränzle et al. for Stochastic SMT extend satisfiability problems with probabilites (Fränzle et al., 2008, Littman et al., 2001). They introduce a probability of satisfiability for SMT formulas with placeholder variables and probability distributions D_S. They define aggregation functions to aggregate probabilities of satisfiability both for variables with probability distributions and for existentially quantified variables without probability distributions (regular variables). If a (sub-) formula is satisfiable, they use its probability value for multiplication. If a formula does not contain probability distributions, that is, it just contains regular existentially quantified variables, they aggregate probabilities of subformulas by selecting the maximum probability value.

However, choosing aggregation functions like a weighted average or maximum value is not trivial. Therefore, we recommend this as a starting point for future work if uncertainty, and therefore probabilities, should be incorporated into the verification framework of this thesis.

6.3 Special Cases

The previous section introduces the idea of including a probability of correctness of a service composition into verification. In sequential compositions, there is exactly one proof obligation, but for service compositions with loops, we have to consider the loop-based proof obligations as well ($\rightarrow$ Definition 5.9, Proof Obligations for Total Correctness, $\rightarrow$ Theorem 5.10, Provability corresponds with Tautology).

6.3.1 Loop Proof Obligations

One way to formalize the behavior of loops is to *unroll* a fixed number k of loop executions, as done by *bounded model-checking* (BMC) approaches (Armando et al., 2009, Biere et al., 1999). While unrolling captures the exact behavior of the loop, it is by definition bound to k steps. Instead of BMC, we capture the behavior of the loop by a *loop invariant* and assume that it is already given for every loop ($\rightarrow$ Definition 5.6, Loops, Invariants, and Termination Functions, $\rightarrow$ Section 5.1, Related Work and the Treatment of Loops). To prove correctness automatically, this results in having additional proof obligations to show termination.

There are two ways to integrate additional proof obligations into the probability of correctness of a service composition. At first, the original proof obligation can be replaced by a *set* of proof obligations that have to hold; the combined obligation only holds, if every obligation in the set holds. At second, the probability of correctness of a composition can be defined *for every proof obligation*. This way, a more fine-grained result can be achieved, and errors of a service composition can be tracked down to, e.g., an invariant that is too weak, or a termination function which does not hold.

6.3.2 Repetition

In the sketch to include uncertainty into verification as described above, every call to a single service S is handled by exactly the same service description variable φ_S°. This reflects that we make a decision whether to use the “correct” or the “faulty” representation of a service *once*. Depending on the circumstances, we may want to have this uncertainty for every occurrence of S *separately*, e.g., if a service is called twice in the same composition. In that case, every occurrence of φ_S° in a workflow template encoding gets a separate probability distribution.

A major drawback of this approach is that loops cannot be handled without further knowledge, as the number of loop executions has to be known in advance. However, it is possible to use techniques from bounded model checking to unroll the loop for a fixed number k of loop iterations.

Whether or not several occurrences of φ_S° should be handled independently, depends on the original source of the probability of correctness. While *computing* a PC value of a service composition works with both versions (with the exception of loops), its *interpretation* is closely related to the original modeling intentions. For example, if the source of PC is related to reputation, it seems reasonable to evaluate every service description variable of the same service S only once; if the source is related to non-functional properties, separate evaluations seem to be more appropriate.

6.4 Discussion and Related Work

In this chapter, we presented an idea to combine uncertainty of service descriptions with a formal verification of service compositions. We do this by providing a generic notion of *probability of correctness* (PC) of service descriptions. As a result we identified the need to further investigate means to aggregate PC values and verification results of all possible workflow template encodings. With such an aggregation method we get a probability of correctness of an overall composition which is backed by concrete verification results. The interpretation of this probability, however, depends heavily on the computation of the original PC values. As an example, we may use simplified *reputation* (or *trust*) *values* (e.g., Erickson, 2009). Other approaches are based on Fuzzy sets (Wu and Weaver, 2006), or provide a configurable framework by itself (Jungmann et al., 2014). Generally, and in contrast to our approach, reputation-based notions of uncertainty have no impact on verification of functional properties.

Another promising source of PC values is the monitoring of violations of described behavior, as done using runtime monitoring (e.g., Wonisch et al., 2013b), where a monitor observes whether a program violates a formally specified property during runtime. Other formal verification techniques, which are combined with probabilistic behavior, typically aim at *quality of service* properties, e.g., the model checker Prism (Gallotti et al., 2008, Kwiatkowska et al., 2011). However, these approaches are built on sound service descriptions, that is, service descriptions are always trusted. Littman et al. and Fränzle et al. combine uncertainty with satisfiability problems in general, but they focus on probabilistic determination of variable values (Fränzle et al., 2008, Littman et al., 2001).

While functional verification has moved from classical program verification (e.g., Lahiri and Qadeer, 2008) to the world of Semantic Web (e.g., Ankolekar et al., 2005), the notion of uncertain service descriptions is, as far as we know, not treated yet in SAT-based verification.

Chapter 7

Prototypical Implementation

This thesis provides two theorems related to satisfiability: Theorem 5.10 (Provability corresponds with Tautology) states the equivalence of correctness of a service composition template with the tautology of a corresponding logical formula, and Theorem 4.24 (Constraint Rule Compliance) states the equivalence of creating a correct instantiation of a service composition template with the compliance of its instantiated contraint rules with the target domain ontology. Both theorems relate central topics of this thesis with satisfiability problems, claiming that tautology of a formula can be solved by existing satisfiability solvers. This chapter introduces a prototypical implementation called Spike to automate these satisfiability checks.

Section 7.1 gives an overview about the overall process of automating satisfiability checks for service composition templates and its relation to the CRC 901 tool set SeSAME. Section 7.2 introduces the SMT-LIB standard to encode satisfiability problems. Section 7.3 presents the architecture of the prototype implementation, and Section 7.4 continues with an evaluation.

7.1 Overview

This section gives an overview on the overall encoding workflow, the relation to SeSAME, and the input of Spike.

7.1.1 Automating Satisfiability Problems

The important practical consequence of the theorems relating correctness to satisfiability of logical formulas is that solving satisfiability of the latter is already supported by tools. While there exist solvers for specialized domains as well as for propositional satisfiability (e.g., MiniSAT/CHAFF, Eén and Sörensson, 2004) or subsets of predicate logic (e.g., MathSAT, Cimatti et al., 2013, Microsoft's Z3, de Moura and Bjørner, 2008), we concentrate on solvers for *satisfiability modulo theories* ($\rightarrow$ Section 7.2). With automated

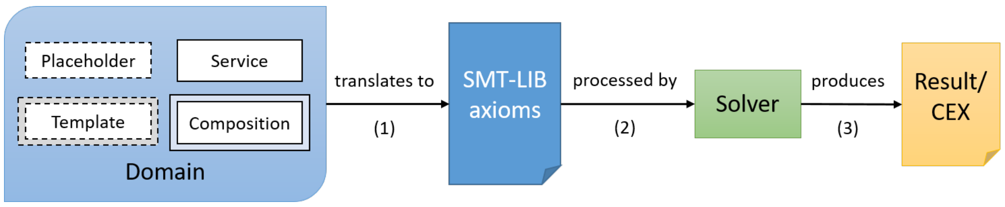

Figure 7.1: Process overview on automated verification

tool support for solving satisfiability problems available, it is sufficient to translate the question of whether or not a composition is correct into a logical formula and then utilize existing solvers (→ Chapter 5). The solver result – a formula is tautological or not – can then be translated back into correctness properties using the theorems. This approach consists of three steps (→ Figure 7.1).

(1) The knowledge base, service descriptions, compositions, and templates, including workflows, have to be translated into their logical representation as defined in Chapter 5. This representation has to be encoded using a language that is compatible with the accepted input of a given solver. For this thesis, we choose SMT-LIB 2.5, a standard language for satisfiability modulo theories (Barrett et al., 2015).[1]

(2) The resulting SMT-LIB representation is used as input to an SMT solver, in our context Microsoft's Z3.[2]

(3) The solver then produces a result, either a successful execution of the proof by showing tautology, or a counterexample.

With the *Service composition Prover with Integration of Knowledge base Encodings* (SPIKE), we provide a prototypical implementation of this process.

7.1.2 Compatibility with SeSAME

In the context of CRC 901 On-The-Fly Computing exists the SESAME tool kit (Arifulina et al., 2014). SESAME is a collection of tools to model services and service compositions and to handle tasks related to these models, e.g., service matching. To this end, SESAME defines a *Service Specification Language* (SSL) based on the *Palladio Component Model* (PCM, Becker et al., 2009). SSL provides language features to model service signatures, pre- and postconditions, and service compositions. Service compositions use *service effect specifications* (SEFFs) as workflow language (Arifulina et al., 2015). An early prototype of SPIKE implementing first steps of the verification process of this thesis exists within the SESAME tool set as part of the *Functional Analysis Tools*

[1]`http://www.smtlib.org`, retrieved July 21, 2017.
[2]`https://github.com/Z3Prover/z3`, retrieved July 21, 2017.

(FAT).[3] However, there exist some incompatibilities between the specification language presented in this thesis and the capabilities of SSL. These incompatibilities are:

(1) SSL is specialized to model service workflows. Consequently, the only atomic workflow element is the service call. It is therefore not possible (it is even contrary to the strategic goal of SSL) to model variable assignments as defined in the assignment statement of the modeling language of this thesis (→ Definitions 3.3, Workflow, and 4.5, Workflow Template).

(2) SSL relies on the Web Ontology Language (OWL) as modeled by Stanford's ontology editor Protégé (Horridge et al., 2014). However, OWL does not provide native support for modeling set-based data types, and SSL does not add additional support. The approach of this thesis handles this limitation by deriving data types from ontologies and not requiring them to be defined solely within the ontological knowledge (→ Section 2.2.5, A Formal Knowledge Base).

(3) SSL supports logical expressions to be used in pre- and postconditions, but the corresponding sublanguage only supports logical and linear arithmetics operands and not set operands. While the language can be extended with set support in principle, this contradicts topic (2).

As a result, the early prototype demonstrates general integration into SeSAME, but is limited to non-set types. Additionally, it does not support loops, as the use of invariants typically relies on relating old and new variable values, which is done by introducing new variables, contradicting topic (1). Therefore, the final prototype of Spike as described in this chapter is not integrated into SeSAME, but developed as a stand-alone application.

7.1.3 Input Models

In SeSAME, SSL is the integrated modeling language to model services and compositions. It refers to OWL with SWRL rules to model ontologies and directly derives the allowed data types from the domain knowledge. In Spike, the data structure for knowledge base modeling directly reflects the knowledge base definition (→ Definition 2.3) and can therefore easily serve as adapter for ontology languages like OWL. Additionally, this extends the capabilities of SWRL by allowing for arbitrary logical expressions as rules, instead of restricting rules to Horn clauses. Service representations consist of a signature with pre- and postconditions, both in SSL, Spike and in the formal definition.

In SSL, modeling of workflows with SEFFs does not follow the relatively free form of UML activity diagrams, but it directly adapts the restricted structure of PCM. This results in a consistent workflow structure where self-contained blocks are central elements. Therefore, nested workflows are easily modeled. For details on SSL, we refer to Arifulina,

[3] `http://sfb901.uni-paderborn.de/sfb-901/projects/tools-demonstration-systems/sesame.html`, retrieved July 21, 2017.

Platenius, Becker, Engels, and Schäfer (2015). Because of the incompatibilities above Spike does not support SSL. Instead, the input model of Spike follows Definition 4.5 (Workflow Template) and therefore uses a block-based structure. This makes a later integration with SeSAME easy. In contrast to SSL, it supports a dedicated variable assignment element, and data types are based on the knowledge base model, but extended with set types.

From a technical point of view, the input model can be used as adapter layer, following the Adapter design pattern (Gamma et al., 1994), to couple the prototype to a future version of SSL (Figure 7.2).

7.2 Logical Models

This section introduces the SMT-LIB standard and dedicated SMT encodings.

7.2.1 Logical Encoding Standard SMT-LIB

Solving a logical formula is the process of finding a model which satisfies the formula, or, an assignment for which a formula evaluates to *true*. This process can be automated, and Davis, Putnam, Logemann and Loveland gave an algorithm to solve propositional formulas already in the 1960s (Davis and Putnam, 1960, Davis et al., 1962). Since then, their algorithm, dubbed as *DPLL*, serves as foundation for several satisfiability (SAT) solvers. Tinelli and Ganzinger et al. extend DPLL by integrating (in principle arbitrary) *theories* (Ganzinger et al., 2004, Tinelli, 2002). This is done by lifting the DPLL calculus from propositional formulas to more complex expressions by changing the definition of literals. In DPLL with Theories, or *DPLL($\mathcal{T}$)*, a literal is an arbitrary expression which can be entailed (or proved to be unsatisfiable) by way of using an additional theory. This approach is the foundation of *satisfiability modulo theories* (SMT).

To leverage the development of solvers, a notational standard for SMT expressions, both for core SMT and a selection of theories, evolved: the SMT-LIB 2.5 standard (Barrett et al., 2015). The standard includes theories like linear arithmetics, equality, and bitvectors. An important theory for the scenario of this thesis is the theory of *uninterpreted functions* (UF). Uninterpreted functions allow for the use of function symbols in logical expressions without fixing an interpretation. As a result, a solver supporting UF tries to

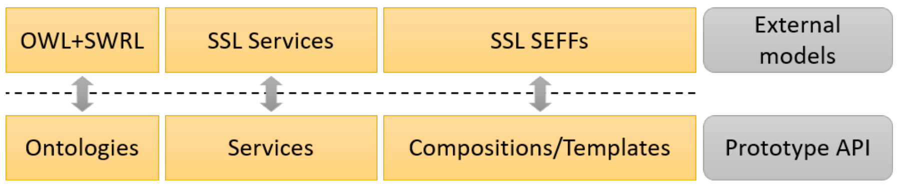

Figure 7.2: Spike provides an API which can be used to adapt external input models

solve not only for a satisfying model in terms of variable evaluation, but additionally in terms of finding an interpretation for said function symbols, which result in a satisfying evaluation of the formula in question. This is done by treating function symbols as literals with regard to standard DPLL, but using the underlying theory of uninterpreted functions to find a model (including an interpretation for the function symbol) which entails the remaining part of the DPLL problem. In practice, this enables us to automatically solve *predicate logic* expressions (combined with linear arithmetic). This leads to a class of problems which is undecidable in general, and while quantifiers are not part of the SMT-LIB standard, some solvers (like Microsoft's Z3) support them to a certain extend. On a practical level, satisfiability of quantified expressions is often solvable.

On a conceptual level, uninterpreted functions, and finding their logical interpretation, provide the link between the domains of ontology modeling (and subsumption checking) and both formal verification and modeling of service compositions ($\rightarrow$ Figure 1.1). Finding interpretations, and therefore logical structures, which satisfy a formula, is at the core of the theoretical foundation of this thesis.

Although dedicated ontology solvers exist, encoding description logics and subsumption problems for SAT solvers does also exist. Sebastiani and Vescovi translated the basic DL $\mathcal{ALC}$ as satisfiability problem and proved the soundness and correctness of their encoding using Kripke structures (Sebastiani and Vescovi, 2006, 2009). Also, they gave an SMT encoding for DLs with quantified role restrictions ($\mathcal{ALCQ}$, Haarslev et al. 2011), and Vescovi discussed the application of SMT to ontology reasoning in general in his PhD thesis (Vescovi, 2011). Core idea is the translation of $\mathcal{ALCQ}$ into SMT with the *theory of costs*, to handle the linear arithmetic part of role quantification. However, these approaches do not provide a general translation for $\mathcal{SHOIN}$ or $s\mathcal{ROIQ}$, which are used to define the semantics of OWL and OWL2 ($\rightarrow$ Chapter 2).

7.2.2 Encoding Proofs, Sets, and Polymorphic Types

The theorems of this thesis rely on the tautology of logical formulas in the form of $\models_K^{CR}$ Req ($\rightarrow$ Definition 5.10). As SMT solvers cannot show tautology directly, in practice any check for tautology is translated into its corresponding check of contradiction, as for every logical formula p holds that if and only if p is a tautology, then its negation $\neg p$ is unsatisfiable. Therefore, every proof obligation which requires tautology to show that a proof exists is negated and a contradiction (or Unsat) is the solver result which indicates a successful proof.

Another core element of our expressions is the use of *sets*. There is no theory of sets for SMT yet, though Kröning, Rümmer, and Weissenbacher (2009) propose a notation, but not a corresponding decision procedure. Instead, de Moura and Bjørner propose to use the *extended theory of arrays* to encode sets of arbitrary size (de Moura and Bjørner, 2009). Formally, a set variable is an array with an element as index and a boolean constant as value, indicating whether or not the element is part of the set. The set encoding of SPIKE is based on this proposal.

A third key element concerns the encoding of a knowledge base. SMT-LIB supports only a monomorphic type system, while our definition of domain knowledge allows for type hierarchies. We solve this issue by using a single unique data type T for scalar types in SMT-LIB and introducing a type predicate for every type in our type system. Andreas Krakau evaluated this approach in his bachelor's thesis (Krakau, 2014). Based on type predicates, he modeled subtype relations using implications. In SPIKE, we do not use implications, but encode subtype relations based on arrays. The subtype relations of a type T_1 are stored in boolean arrays with the unique ID of the potential supertype as index, effectively providing a lookup table.

7.3 Prototype Architecture

The prototype implementation of SPIKE follows a layered architecture (Figure 7.3). The top layer consists of an application programming interface (API) to specify the *input models* for knowledge bases, service descriptions, and service composition specifications, including templates and workflows. Knowledge base modeling follows closely the definition in Chapter 2, providing means to handle concepts, roles, and rules. Set types are included, but optional. Service descriptions also follow the pattern of signatures (including typed input and output parameters) with pre- and postconditions. Service compositions and templates re-use service descriptions with additional workflows. Workflow specifications follow their formal definition.

The prototype implementation layer consists of the following parts:

- The *API layer* provides an interface to define knowledge base models, service definitions, and compositions and templates. The API can be used directly by, e.g., test scripts or a graphical user interface, or by serving as a backend for adapters to, e.g., SESAME.
- The model created by the API layer is decoupled from the original input model (e.g., OWL or SSL) to value the principle of separation of concerns. It closely follows the formal definition as defined in this thesis.
- The *code generation modules* generate SMT-LIB axioms for their associated data models.
- The *solver control module* connects to the API of the actual SMT solver. We use Microsoft's Z3 because of its support of quantification.
- The *statistics* module collects runtime information.

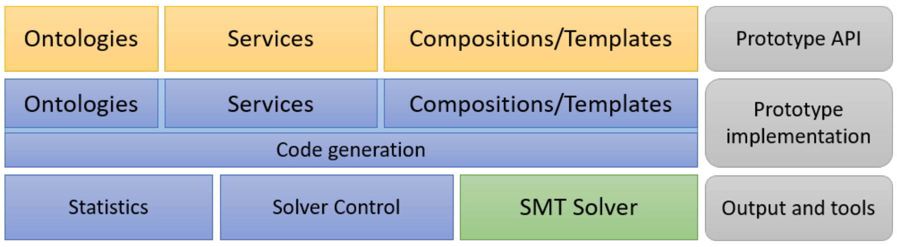

FIGURE 7.3: SPIKE implementation layers

Without a SESAME integration, input models have to be defined programmatically. The input data model provides an intuitive API oriented on the definitions in this thesis and similar to the Z3 Python API. Due to the separation of input data and the use of an external solver, the prototype can easily be integrated with other tools.

- As mentioned before, integration into SESAME by utilizing the structural similarity of the input data model to SSL SEFFs is straightforward, as soon as SESAME supports set modeling and variable assignment actions.
- An example of an additional postprocessing step is the *counterexample visualization*, which translates the satisfying assignment of a failed proof (the counterexample) into a graph-based visualization. The counterexample visualization is part of the Functional Analysis Tools of SESAME, but works on satisfying assignments in general and was implemented by Siddharta Moitra.[4]

Being developed as a set of Python scripts, the prototype has no dependencies to external frameworks apart from the Z3 solver and can be directly re-used and modified. It is available on Bitbucket by request.[5]

[4]His implementation is part of the SESAME Functional Analysis Tools: `http://sfb901.uni-paderborn.de/sfb-901/projects/tools-demonstration-systems/sesame.html`, retrieved July 21, 2017.

[5]`https://bitbucket.org/swalther/spike`, as of July 21, 2017.

7.4 Evaluation

The SPIKE prototype is a proof of concept to show the following:

(1) One of the core contributions is the correspondence of correctness of a template to the satisfiability of a logical formula (contribution 3, p. 9). Using the logical encoding of a correctness check of a template as given in Chapter 5 works in general.

(2) Another core contribution is the ability to check the instantiated template constraints instead of the complete composition (contribution 2, p. 2). The check of constraint instantiation is especially useful if a template cannot be verified automatically.

As a service market in the understanding of the vision of the CRC 901 does not exist yet, especially in the level of formality as required by this thesis, we do not do a quantitative evaluation. Instead, we use SPIKE to answer the given questions exemplarily. In the following, we give an overview about the evaluation approach and draw a conclusion.

7.4.1 Approach

The overall process of creating a service composition from a service composition template consists of two main steps related to correctness: verifying a template and instantiating it as an actual composition. SPIKE implements the logical encoding as formalized in Chapter 5. We use SPIKE

(1) to verify exemplary templates,

(2) to execute constraint checks,

(3) and to demonstrate how a failed constraint check can be utilized to correct an instantiation.

The following templates represent simple patterns that can be used for structurally similar tasks in different domains. The template names reflect a typical usage pattern. All templates use sequences and service calls in addition to complex statements. Table 7.1 gives an overview of the templates; the complete definitions are listed in the appendix. The ontologies used for the templates are small and contain only the concepts and roles which are necessary for template definition. The key functionality of the template is defined by constraint rules, relating postconditions of service placeholders to roles (or properties) of the ontology. The FILTER template uses the example ontology from example 4.3, which we repeat here:

TABLE 7.1: Service composition template examples

Template	**Complex Stmts**	**Purpose**	**Page**
Produce/Consume		Creates intermediate data based on some input, which is consumed again for some output	168
Choose	Conditional	Selects one of two inputs based on the result of a service call on both of the inputs	169
Target Processing	While-loop, Conditional	Processes an input until a given target property is reached; after that, uses the processed value as output	170
Filter	Foreach-loop, Conditional	Selects a subset of an input set as the output, based on a property that is obtained using a service call	171

$$\begin{aligned}
C_T &= \{\textsc{Element}, \textsc{Value}\}\\
P_T &= \{\textsc{hasValue} : \textsc{Element} \times \textsc{Value}\ ,\\
&\qquad \textsc{isTargetElement} : \textsc{Element} \times \textsc{Bool}\ ,\\
&\qquad \textsc{isTargetValue} : \textsc{Value} \times \textsc{Bool}\}\\
\sqsubseteq_T &= \{\}\\
R_T &= \{\mathit{functional}(\textsc{hasValue}), \mathit{functional}(\textsc{isTargetElement}),\\
&\qquad \mathit{functional}(\textsc{isTargetValue})\}
\end{aligned}$$

The ontologies needed to verify the other templates are of comparable complexity and can be found in the appendix at the description of the respective template.

7.4.2 Results

To verify templates, their pre- and postconditions serve as requirements. All templates have been automatically verified against their pre- and postcondition, with the exception of the FILTER template. Table 7.2 shows the overall runtime of Z3. We use the mean of 100 runs (rounded to milliseconds), executed on an Intel Atom x7-Z8700 with 1.6 GHz with Windows 10 and 4 GB RAM, using Z3 version 4.5.0-x86.

For the FILTER template, the solver cannot find a solution and returns “unknown” after the indicated time. To analyze the cause of this result, we break down the logical encoding of the overall correctness formula into sub-formulas, as it consists of a conjunction of different verification conditions ($\rightarrow$ Definitions 5.8 and 5.9), namely one representing the requirements and the overall workflow and more for every loop. If we check tautology for all of them separately, we find that $VC(\mathbf{inv} \wedge B, W, \mathbf{inv})$ cannot be solved by the solver (with W the loop body, $\mathbf{inv}$ the invariant, and B the loop condition). For

TABLE 7.2: Solver runtimes and results

Template	Complex Stmts	Result	Z3 Rt [ms]
Produce/Consume		proved	12
Choose	Conditional	proved	14
Target Processing	While-loop, Conditional	proved	12
Filter	For-loop, Conditional	unknown	9,010

`foreach` loops, the "condition" to enter the loop is that there are still elements in the loop set (→ semantics of loops in Def. 4.7), that is, $A \neq \emptyset$ with A being the loop set. With the current set encoding, this cannot be proven, as the extended theory of arrays is based on unbounded arrays (de Moura and Bjørner, 2009).

From this we learn two things:

- If we are able to prove the invariant manually for `foreach` loops, proving overall correctness may still be possible. This is especially true for templates where loops are only a small part of the overall workflow.
- We chose the extended theory of arrays deliberately. If we set decidability first, we could use an alternative encoding, e.g., using bit vectors. While the general approach of encoding set membership does not change, proofs only hold true for a given maximum size of sets and may become false for larger sets.

As we proved the FILTER template to be correct manually (→ Figure 4.3), we are still able to utilize the constraint check to verify the correctness of an actual composition. Here, we replace service placeholders with actual services from the domain of tourism (→ Example 2.1) for the following example instantiations. In both examples, the constraints of the template are translated to the target ontology, using the service's pre- and postcondition instead of the service placeholder predicates. As these constraints have to be implied by the knowledge base, we use the same technique as for template verification. We add the negated constraints to the knowledge base and check for contradiction.

Example 7.1 (Correct FILTER Instantiation)**.** *For easier reading, we give a tabular instantiation view.*

	Template Ontology	Domain Ontology
Concepts	ELEMENT	RESTAURANT
	VALUE	RATING
Roles	ISTARGETELEMENT	GOODRESTAURANT
	ISTARGETVALUE	ISMINRATING
	HASVALUE	ISRATING
Service	*Acquire*	*GetRating*

The service GetRating *has the following signature and pre- and postcondition (→ Example 3.1):*

$$\begin{aligned}&\{\}\\ &\mathit{GetRating}(\mathit{restaurant} : \mathit{Restaurant}, \mathit{rating} : \mathit{Rating})\\ &\{\mathit{hasRating}(\mathit{restaurant}, \mathit{rating})\}\ .\end{aligned}$$

In the first example, the check succeeded: The instantiation yields a correct composition according to Theorem 4.24 (Constraint Rule Compliance).

Example 7.2 (Wrong Filter Instantiation)**.** *Do demonstrate a wrong instantiation, we use the same mappings as in the example above, but we map to a slightly different ontology. We also use the Tourism domain, but we do not assume that a "good restaurant" is in any way related to a formal rating, that is, we omit the following rules from the ontology on page 24:*

$$\begin{aligned}&\textsc{hasRating}(\mathit{res}, \mathit{rat}) \wedge \textsc{isMinRating}(\mathit{rat}) \Rightarrow \textsc{goodRestaurant}(\mathit{res}),\\ &\textsc{hasRating}(\mathit{res}, \mathit{rat}) \wedge \neg\textsc{isMinRating}(\mathit{rat}) \Rightarrow \neg\textsc{goodRestaurant}(\mathit{res})\end{aligned}$$

In the second example, the instantiated constraints cannot be concluded from the knowledge base. The satisfying model generated by the solver indicates variable assignments and predicate interpretations such that the domain knowledge is respected, but the instantiated constraints are violated (because their negations are satisfied). For illustration, we simplified the satisfying model; the complete model is part of the appendix on page 173.

Example 7.3 (Satisfying Model for Template Analysis)**.** *When analyzing a satisfying model representing a counterexample, we first look at predicate interpretations which are related to violated instantiated constraints. Here, the instantiation of constraint predicates are* goodRestaurant *and* isMinRating*. The satisfying model produces very simple interpretations for these predicates:*

$$\begin{aligned}\textsc{isMinRating}(r) &= \top\\ \textsc{goodRestaurant}(r) &= \bot\end{aligned}$$

Therefore, regardless of the variable assignment, we can have restaurants with minimal ratings which are not necessarily good. This aligns with the intention of the knowledge base, but violates the constraints of the instantiated template.

The analysis of template verification counterexamples works the same way.

7.4.3 Conclusion

We have shown that the automatic verification of service composition templates by means of solving a logical formula as proposed in Chapter 5 works in principle. However, the use of sets in combination with `foreach` loops can already lead to unsolvable formulas. In this case, we can prove the template manually to be correct and apply the constraint check also proposed in Chapter 5. Using this check, we need to check satisfiability not of the complete template representation, but of the instantiated template constraints in the target ontology. This worked in the given example. We also demonstrated the use of counterexamples.

While the templates are generic and can be applied in various domains, their specification detail in terms of formula size is quite small. We therefore propose to re-evaluate the applicability of automatic verification as soon as a larger set of formally specified services and templates is available.

Part III

Discussion

Chapter 8

Discussion and Conclusion

This chapter discusses approaches related to the contributions of this thesis, either in providing an integrated verification framework, dealing with correctness by construction similar to our service composition instance checks, automating verification, or combinations thereof. After the discussion, the chapter provides a concise summary of our core contributions, and it concludes with some remarks about possible future work.

8.1 Related Approaches

The core contributions of this thesis are threefold: We provide a formal proof framework spanning the domains of knowledge representation, service and workflow modeling, and program verification; we ease the task of verifying service compositions based on templates by reducing the verification task to a knowledge representation check; and we provide a logical representation of the correctness properties of templates to leverage automatic verification. These contributions lie at the very heart of these domains, and combining the three different research topics is an important key characteristic. However, a lot of approaches are concerned with one or two of the topics. The different chapters of this thesis discuss related work in the context of their respective focus; now, we discuss some approaches which come particularly close to some of the contributions of this thesis.

At first, we focus on formal correctness of programs and means to automatically prove it. Theoretical approaches date back to Hoare (Hoare, 1969) and Floyd (Floyd, 1967). Especially approaches like CSP (Hoare, 1978, 1985) or the B method (Abrial et al., 1991) are based on defining programs by refining their formal specifications, thus following a correctness-by-construction principle. In practice, functional programming languages follow a similar paradigm, as they define the problem to be solved logically and not the actual actions needed to solve it. The *Design-by-Contract* principle and the Eiffel language brought the idea of pre- and postconditions and invariants to the domain of imperative and object-oriented, non-functional programming (Meyer, 1992, 1997). The formal definition properties of imperative programs, or – taking a modular viewpoint – single methods are part of the basic foundation of automatic verification of programs. Other approaches like *JML* (Java Modeling Language, Leavens et al., 1999) add those annotation capabilities to Java and pave the way for static analyses as, e.g., the *Extended Static Checker for Java* (ESC/Java, Cok and Kiniry, 2005, Flanagan et al., 2002).

Languages to specify annotations to method calls, programs, or services are one prerequisite of automatic verification; tool support is the other. With the progress in the development of SAT and SMT solvers and standards like SMT-LIB2 (Barrett et al., 2015), existing SMT solvers like Z3 (de Moura and Bjørner, 2008), MathSAT (Cimatti et al., 2013), or CVC (Barrett et al., 2011) are capable of providing a backend to automate current verification techniques. Tools like Dafny (Leino and M., 2010) and VeriFast (Jacobs et al., 2011) are built on top of SMT solvers to ease the modeling of SAT/SMT problems. As SMT solvers are not dedicated verification tools, they serve as foundational tooling for other logic-related tasks, from inferring program and loop invariants, weakest preconditions and strongest postconditions (e.g., Gulwani et al., 2008) to program synthesis based on formal specifications (e.g., Srivastava et al., 2010).

In the following, we discuss two approaches which define *intermediate verification languages* as an intermediate step between specification of a program's (or method's) properties and an SMT solver's input language.

The *Why3* approach addresses the issue of providing automatic verification independently of both the specification or programming language to be verified and the underlying theorem provers or solvers (Filliâtre, 2013, Filliâtre and Paskevich, 2013). To this end, Why3 provides a logic-based semantics for its specification language *WhyML*, which is self-contained. WhyML can be used to formalize conditions and properties of programming languages, which are then used to generate logical verification conditions which can serve as input to automatic solvers or theorem provers. WhyML, e.g., serves as logical language for proof obligations generated by the verifier *Krakatoa* (Marché et al., 2004), which verifies Java programs annotated with JML. As Why3 is generic and language-independent, it serves as an intermediate verification language and framework. While this is its strength, it also means that using it for a concrete language always means close inspection of the compatibility of its semantics with the semantics of the verified language.

Another framework, which is tightly integrated with a modern programming language, is *Spec#* (Barnett et al., 2005). Drawing inspirations from Eiffel and JML, it adds new language constructs for formal specification to Microsoft's popular C# programming language, with the ability to define pre- and postconditions for methods, or *method contracts*, as the main feature. Spec# can be used in two ways: During runtime, the .NET framework (which compiles and executes .NET based languages like C# just as a Java Runtime Environment does for Java) generates runtime checks to provide runtime monitoring of the specified pre- and postconditions. If the conditions of a method are violated during a given run of the program, the framework throws an exception.

Instead of runtime monitoring, the static checker *Boogie* (Barnett et al., 2006) can be used to verify that the conditions cannot be violated during a program run, if possible. In contrast to Why3, Spec# builds directly on the C# programming language. While this ensures a practically relevant context, there is no formal semantics of the C# programming language and therefore no formally defined semantics in the sense of this thesis.

From a direction of verification of templates and creating correct instances, there is a close connection to refinement approaches like CSP or B on the one hand, and to any approach built on modular verification on the other hand, as templates in general can be seen as a specific use case of modularity: In an on-the-fly context, a template differs from a service composition only in the sense that it does not use existing services but service placeholders. We highlight three approaches, coming from different directions, which have similarities to our treatment of templates and correctness.

While the use of templates is common in software design – either explicitly by dedicated modeling constructs or implicitly by applying design patterns –, verification of templates is not necessarily covered. *Compositional model checking* deals with the verification of modular systems, as in, e.g., Grumberg and Long (1991). While they use temporal logic specifications and Moore automata, the approach is based on refinement. Conceptually closer to the template specifications and instantiation checks of this thesis is the CARE approach (Hemer and Lindsay, 1996, 1997). CARE follows a refinement-based approach of modeling using the Z modeling language (Woodcock and Davies, 1996), and generates correct service compositions. In CARE, *fragments* are used for modeling: *primitives* with black-box descriptions, which are proven to be correct externally; and *composites*, which are used to model complex algorithms. Based on the multi-set theoretic semantics of Z, several proof obligations can be derived from specification of a composition. The CARE approach also supports a notion of templates, where parts of the specification – types and primitive fragments – are represented by parameters, that is, specifications without corresponding implementations (Hemer and Lindsay, 1997). This way, the concrete implementation can be instantiated later, as long as it meets the formal description used in the composition. In contrast to CARE, we define correctness for an *incomplete* (not instantiated) template. Once proven correct, we show that it is sufficient to proof that an instantiation adheres to some constraints, while in CARE, proof obligations derived from a final instantiation always contain a complete proof for the complete instantiation.

Other approaches of modular analysis are rooted in the domain of software product lines. Soleimanifard and Gurov (2015) define a language with pointer datatypes to model *code variability*. They do this by defining pre- and postconditions of code sections and method calls which may be replaced by different code (or calls), which has to oblige the pre- and postconditions defined before. This approach, though targeted at code variant analysis and working on temporal safety properties, is conceptually similar to template verification. Another approach addresses templates from a direction of defining concrete program instances by variants of a “core” program. Hähnle and Schaefer (2012) base their approach of variant verification on delta programming, where the core program gets verified and concrete programs can be proved to be correct by analyzing the difference, or delta, in comparison to the core.

Both approaches provide interesting views on template verification, Soleimanifard and Gurov's from a perspective of product line analysis, and Hähnle and Schaefer's from a perspective of refinement and piece-wise analysis. However, both share a missing core feature with the previously discussed approaches: They lack the connection to a formally specified knowledge base.

In the Introduction, we identified three core requirements in the context of verification in an on-the-fly scenario:

(1) A means to formally specify the behavior of services;

(2) Formalized domain knowledge as a foundation for service specifications, both to provide the available vocabulary (predicates) and additional restrictions on its interpretation;

(3) Combination of services into service compositions.

To address these requirements, this thesis provides the three core contributions of an integrated verification framework, the creation of correct compositions based on verified templates, and the use of SMT solvers by giving a logical representation of verification conditions.

Some of the approaches provide similar contributions, especially Why3 with a generic verification framework including a formally defined semantics and its independence of concrete solvers. However, its generality also requires additional adaptation to integrate it into an on-the-fly context, including the application to a service (and workflow) specification language. Spec#, on the other hand, is tightly integrated with a programming language (and thus workflow language), but integration with a formal knowledge base is not straightforward, at it is not based on logic – in contrast to Why3. The software variability approaches have similar characteristics, as the connection of a knowledge model is not common in program verification.

Summarizing, it shows that none of the related approaches completely addresses all requirements of verification in an on-the-fly scenario, expecially requirement (2).

8.2 Conclusion

The paradigm of On-The-Fly Computing provides a vision of a flexible, heterogenous market of software services, where a user does not buy off-the-shelf software, but highly specific, personalized software services tailored to her needs on demand. This market includes the infrastructure for an economic market, for search and combination of services,

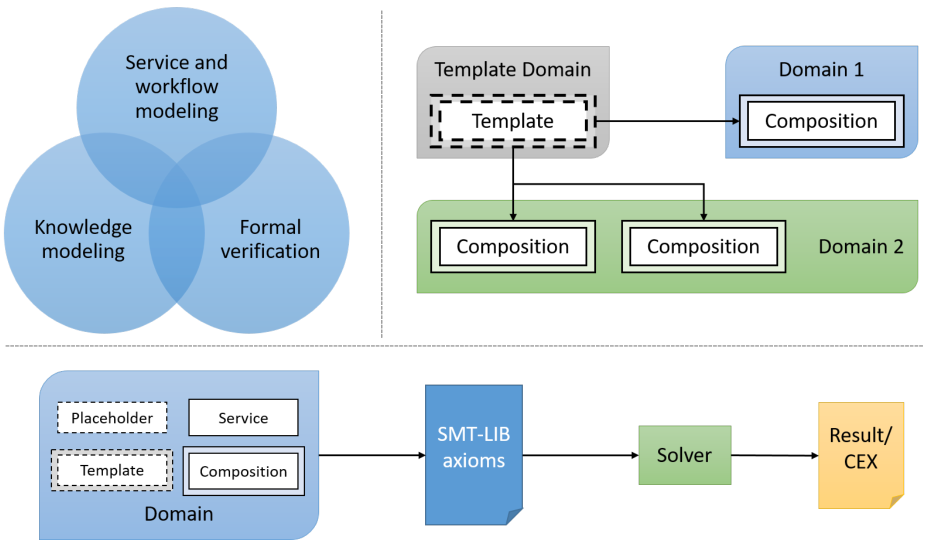

FIGURE 8.1: The core contributions: Theoretical framework (top left), reduction of composition verification to constraint checks (top right), automated verification (below)

and their deployment and execution. Mandatory ingredients to enable such a market are, among others, (1) a formalized knowledge base of the domain(s) of application, (2) a method to formally specify functional (and non-functional) behavior of software services based on this knowledge base, and (3) the techniques to create and deliver compositional services based on the requirements of a user (→ Introduction).

Part of this vision is the goal to deliver only *correct* services to the request of the user: To this end, *formal verification* is to be employed to ensure that an automatically generated service composition matches the original requirements. To provide a framework for formal verification in an on-the-fly context, three research topics have to be addressed: knowledge base modeling; modeling of services, compositions and workflows; and program verification. As these aspects are research topics of their own and typically handled in isolation or in pairs, they provide their own sets of formalisms and, especially, semantics. The challenge to combine them therefore includes to ensure that their modeling formalisms and semantics are compatible, or, ideally, rooted in the same framework.

If these apsects are combined in a pragmatic way only, we cannot draw a reliable conclusion from, e.g., verification results of a service composition. The reason is that the semantics may not be compatible, e.g., the domain knowledge may be modeled by a taxonomy graph, service descriptions by syntactical signatures and accompanying API texts, and compositions by UML activity diagrams. Therefore, in order to make verification results usable, a shared theoretical basis for all three aspects is mandatory.

Figure 8.1 gives an overview about the three core contributions of this thesis. The first core contribution addresses the challenge of a shared theoretical framework and combines knowledge base modeling, service and service composition descriptions (including workflows), and verification by defining their semantics on a common ground. We do this by starting with a knowledge base model based on logical structures (→ Chapter 2). While this is common, we then define service (and service requirements) descriptions based on the resulting vocabulary and therefore give their semantics a formal root in knowledge modeling. Additionally, we parameterize the operational semantics of workflows as used in service compositions with logical structures, whose interpretations are restricted by the knowledge base. A definition of correctness and a sound and complete Hoare-style proof calculus based on the same parameters complement the framework (→ Chapter 3).

The second core contribution takes another aspect of on-the-fly scenarios into account: As the process of creating service compositions is an active research topic by itself, and creating arbitrary compositions automatically is not solved, we address the use of composition *templates*. With templates, the overall structure of a composition can be defined by domain experts, with service *placeholders* instead of existing services. A template, however, can already be verified when it is constructed. To this end, we introduce additional constraint rules for templates, which enable a proof of templates. As a result, when a service composition is created by instantiating a template, its correctness can be shown by showing that the constraint rules can be entailed from the domain of the service composition (→ Chapter 4).

The third core contribution aims at leveraging an automatic verification. The semantics of services and service compositions is rooted in a common logical framework. Based on this framework, we define their logical representation and a correspondence of a correct service composition (and template) with the satisfiability of a logical formula. While this representation is based on predicate logic, and therefore undecidable in general, existing solvers are often able to deal with practical examples (→ Chapter 5). A prototypical implementation puts automatic verification into practice (→ Chapter 7).

Summarizing, we provide a solid theoretical framework to integrate all three relevant aspects of functional verification in an on-the-fly context as well as providing two approaches to ease the verification process.

8.3 Design Decisions

During the development of this thesis, especially in the first phase, several design decisions where made. Starting with the requirements of an on-the-fly context and the focus of functional verification based on pre- and postconditions, utilizing satisfiability solvers to target an automatic verification (in contrast to the use of interactive theorem provers) was the first decision. After experiments with propositional logic and the SAT solver Alloy (Jackson, 2002) it became clear that in the context of domain knowledge modeling the use of predicates and their relations is mandatory. While *descriptions logics* provide this expressiveness and keep decidability at the same time (→ Chapter 2), we decided against them and chose first-order logic and, as a modeling language, SMT-LIB. The rationale behind this decision was to avoid restrictions on the expressiveness of the automation part of the thesis, as this would have restricted the logical formulas we allow to specify services. In practice, decidability now depends on the complexity of the actual domain knowledge and the service specifications, and is not a priori restricted by our formalism.

Using SMT was a decision about the automation of verification of service compositions and templates. As a *modeling* language of those compositions we chose the While language of Apt et al. (2009) over service effect specifications (SEFFs, Becker et al., 2009), prominent programming languages, and other visual modeling approaches. The reasons are that the While language is well understood, comes with a formal semantics based on logical structures, and easily supports formal proofs.

Both SMT-LIB and While are standards or at least well understood languages. To represent domain knowledge, or ontologies, there exist several well-researched modeling and reasoning approaches, too ($\rightarrow$ Chapter 2). However, we chose to define our own ontology language instead of using OWL (Patel-Schneider et al., 2012) or a description language like $s\mathcal{ROIQ}$, which is the formal foundation of OWL2. The main goal of this thesis is the creation of a framework for service composition verification. Automation, and decidability, is a secondary goal. To provide a generic framework, expressiveness of our formalisms is more important than decidability, and as the decision for an actual domain modeling approach like $s\mathcal{ROIQ}$ would include decidability, and therefore restrictions in terms of expressiveness, we decided against it. Instead, our definition of knowledge bases leaves all degrees of freedom, and if decidability is an issue, additional restrictions can keep a knowledge base decidable.

In our first publication we proposed a logical encoding that is based on the input and output of statements (Walther and Wehrheim, 2013). Here, every statement was represented by an encoding with a dedicated set of input and output variables, similar to service calls. For the thesis, we generalized this approach and switched to a state based encoding, where every state of the workflow is identified by its own set of variables. The main difference is that in the thesis, a state l is always characterized by its own, l-labeled set of variable versions for *all* variables. In the previous approach, only variables actually used in an input or output set were covered, which makes data flow modeling of complex workflows tedious at least. Additionally, invariants would be treated as having inputs and outputs as well, which works on a theoretical level, but finding invariants that can be modeled as I/O block in practice is counter-intuitive.

To avoid the encoding of invariants would have been another option. Instead, unrolling loops for a given number of iterations, as in bounded model checking approaches (Biere et al., 1999), is an alternative encoding. However, as the goal of the framework is provability, invariants fit easily into the scheme of Hoare-style proof rules, while providing correctness proofs without restriction to a number of loop unrollings.

8.4 Future Work

We see three main directions for future work. At first, the automatic verification is based on predicate logic. While this makes satisfiability problems undecidable in principle, we believe that – on the scale of on-the-fly service compositions – many problems will be practically solvable. However, as we learned from the evaluation, already the use of sets may force us to make concessions. Further investigations may include

- the use of an alternative SMT-LIB set encoding, e.g., using bitvectors to model sets of a finite size. This solves the problem of undecidable encodings for `foreach` loops, and may be combined with an iterative approach to find a suitable size of the sets;
- the combination of the approach with invariant generation techniques to automatically find invariants based on the loop body;
- the use of bounded model checking to replace invariants. This introduces a flexibility both if invariants may lead to undecidable formulas, and if invariants are are not present to begin with.

On-the-fly service markets and service composition is still a vision in progress, so the claim of solvability has to be analyzed more closely. Additionally, it remains an open issue whether or not a completely formal service description will become accepted. This will probably rely on automatic generation of formal description from informal ones, with automated or semi-automated support. From a computational point of view, formal service descriptions may also be categorized into complexity classes of decidability to leverage appropriate encoding techniques.

At second, the workflow language used in this thesis includes typical workflow statements, but it is not a language used in practice. To make the framework usable, concrete languages – like SSL SEFFs or OWL-S – have to be mapped explicitly to it. To do this, it is important to ensure the compatibility of the underlying semantics.

At third, an on-the-fly context does not only model business software services, but also platform and infrastructure services to provide runtime environments, both hard- and software, for the service compositions of this thesis. Modeling those services layers and integrating them into this theoretical framework would enable a more detailed reasoning in the OTF domain.

Part IV

Appendix

Appendix A

Template Examples

This chapter summarizes the template examples used in the evaluation part of the thesis.

A.1 Produce/Consume

The ontology $K_T = (C_T, P_T, R_T)$ has three concepts, one predicate, and no custom rules:

$$\begin{aligned} C_T &= \{\textsc{Plan}, \textsc{Product}, \textsc{Profit}\}, \\ P_T &= \{\textsc{isHigh} : \textsc{Profit} \times \textsc{Bool}\} \\ R_T &= \{\} \ . \end{aligned}$$

The template accepts one input and produces one output. It acquires a result value for the given input plan ("produce"). This input is then traded for some profit ("consume"). It uses two service placeholders, *Produce* : PLAN $\rightarrow$ PRODUCT and *Consume* : PRODUCT $\rightarrow$ PROFIT.

Template	PRODUCE/CONSUME
Inputs	$x : \mathit{Plan}$
Outputs	$z : \mathit{Profit}$
Precond.	$\mathit{pre}_{\mathit{Produce}}(x)$
Postcond.	$\mathit{isHigh}(z)$
Constraints	$\forall a, b : \mathit{post}_{\mathit{Produce}}(a, b) \Rightarrow \mathit{pre}_{\mathit{Consume}}(b)$ $\forall a, b : \mathit{post}_{\mathit{Consume}}(a, b) \Rightarrow \mathit{isHigh}(b)$

Workflow

```
$y := \mathit{Produce}(x);$
$z := \mathit{Consume}(y);$
```

FIGURE A.1: The PRODUCE/CONSUME template: Produces an abstract product and consumes it for an abstract profit

A.2 Choose

The ontology $K_T = (C_T, P_T, R_T)$ has two concepts, two predicates, and some custom rules:

$$\begin{aligned}
C_T &= \{\textsc{Element}, \textsc{Property}\},\\
P_T &= \{\textsc{isEBetter} : \textsc{Element} \times \textsc{Element},\\
&\quad\ \textsc{isPBetter} : \textsc{Property} \times \textsc{Property}\}\\
R_T &= \{\mathit{irreflexive}(\textsc{isEBetter}), \mathit{antisymmetric}(\textsc{isEBetter}),\\
&\quad\ \mathit{transitive}(\textsc{isEBetter}), \mathit{trichotomous}(\textsc{isEBetter}),\\
&\quad\ \mathit{irreflexive}(\textsc{isPBetter}), \mathit{antisymmetric}(\textsc{isPBetter}),\\
&\quad\ \mathit{transitive}(\textsc{isPBetter}), \mathit{trichotomous}(\textsc{isPBetter})\}\ .
\end{aligned}$$

The template accepts two inputs and produces one output. It selects one of the two inputs based on a quality predicate. It uses one service placeholder, *Fetch* : ELEMENT $\rightarrow$ PROPERTY, to access this predicate.

Template	CHOOSE
Inputs	x, y : *Element*
Outputs	z : *Element*
Precond.	$\mathit{pre}_{\mathit{Fetch}}(x) \wedge \mathit{pre}_{\mathit{Fetch}}(y)$
Postcond.	$\mathit{isEBetter}(x, y) \wedge z = x \vee z = y$
Constraints	$\forall i_1, i_2, k_1, k_2 : \mathit{post}_{\mathit{Fetch}}(i_1, k_1) \wedge \mathit{post}_{\mathit{Fetch}}(i_2, k_2) \wedge \mathit{isPBetter}(k_1, k_2)$ $\Rightarrow \mathit{isEBetter}(i_1, i_2)$

Workflow

```
$u := \mathit{Fetch}(x)$;
$v := \mathit{Fetch}(y)$;
if $\mathit{isPBetter}(u, v)$ then
    $z := x$;
else
    $z := y$;
fi
```

FIGURE A.2: The CHOOSE template: Selects the better of two inputs

A.3 Target Processing

The ontology $K_T = (C_T, P_T, R_T)$ has one concept, one predicate, and some custom rules:

$$\begin{aligned} C_T &= \{T\}, \\ P_T &= \{ <: T \times T\}, \\ R_T &= \{\mathit{irreflexive}(<), \mathit{antisymmetric}(<), \\ &\qquad \mathit{transitive}(<), \mathit{trichotomous}(<), \\ &\qquad \exists b : \forall x : b < x\}\ , \end{aligned}$$

Here, $<$ has a lower bound b.

The template accepts two inputs, a value x and a target value t (according to the comparison predicate), and produces one output. It uses one service placeholder *Process*, which refines the input values by some property which can be compared using the $<$ predicate. The input of the template is processed until the target value is (at least) met. The invariant of the loop is $\mathit{target} < \mathit{output} \vee \mathit{target} = \mathit{output}$, the termination expression is *output*. Here, the type of *output* is not Integer, but T. However, $<$ and $=$ induce an order on T. It uses a service $\mathit{Process} : T \times T \to T$.

Template	TargetProcessing
Inputs	$\mathit{input}, \mathit{target} : T$
Outputs	$\mathit{output} : T$
Precond.	$\mathit{target} < \mathit{input}$
Postcond.	$(\mathit{output} = \mathit{target} \vee \mathit{output} < \mathit{target})$
Constraints	$\forall x, y, z : \mathit{post}_{\mathit{Process}}(x, y, z) \Rightarrow z < x$ $\forall x, y : y < x \Rightarrow \mathit{pre}_{\mathit{Process}}(x, y)$

Workflow

```
$output = input$;
while $target < output$ do
    $output := Process(output, target)$;
od
```

Figure A.3: The TargetProcessing template: Processes its input until a target value is reached

A.4 Filter

The ontology $K_T = (C_T, P_T, R_T)$ has two concepts, three predicates, and three custom rules:

$$\begin{aligned}
C_T &= \{\textsc{Element}, \textsc{Value}\} \\
P_T &= \{\textsc{hasValue} : \textsc{Element} \times \textsc{Value}\ , \\
&\qquad \textsc{isTargetElement} : \textsc{Element} \times \textsc{Bool}\ , \\
&\qquad \textsc{isTargetValue} : \textsc{Value} \times \textsc{Bool}\} \\
\sqsubseteq_T &= \{\} \\
R_T &= \{\mathit{functional}(\textsc{hasValue}), \mathit{functional}(\textsc{isTargetElement}), \\
&\qquad \mathit{functional}(\textsc{isTargetValue})\}
\end{aligned}$$

The template filters a list of inputs by a predicate which has to be obtained using a service *Acquire* : Element → Value. Figure 4.3 shows the proof of correctness.

Template	Filter
Inputs	A : **set** Element
Outputs	B : **set** Element
Precond.	$\forall a \in A : \mathit{pre}_{\mathit{Acquire}}(a)$
Postcond.	$B = \{b \in A \mid \mathit{isTargetElement}(b)\}$
Constraints	$\forall x, y : \mathit{post}_{\mathit{Acquire}}(x, y) \wedge \mathit{isTargetValue}(y) \Rightarrow \mathit{isTargetElement}(x)$ $\forall x, y : \mathit{post}_{\mathit{Acquire}}(x, y) \wedge \neg\mathit{isTargetValue}(y) \Rightarrow \neg\mathit{isTargetElement}(x)$

Workflow

```
$Z := A$;
$B := \emptyset$ ;
foreach $z \in Z$ do
    $y := Acquire(z)$ ;
    if $isTargetValue(y)$ then
        $B := B \cup \{z\}$
    else
        skip
    fi
od
```

Figure A.4: The Filter template to filter a set

Appendix B

Listings

B.1 Counterexample

Counterexamples for automated proofs are variable assignments and predicate interpretations that hint to specification errors. On page 153, a counterexample for a failed proof of the FILTER template is discussed in shortened form. This is the complete output of the Z3 solver for the failed proof.

```
;; universe for T:
;;   T!val!12 T!val!6 T!val!8 T!val!5 T!val!10 T!val!11 T!val!3 T!val!14 T!val!0
    T!val!4 T!val!2 T!val!13 T!val!15 T!val!9 T!val!7 T!val!1
;; -----------
;; definitions for universe elements:
(declare-fun T!val!12 () T)
(declare-fun T!val!6 () T)
(declare-fun T!val!8 () T)
(declare-fun T!val!5 () T)
(declare-fun T!val!10 () T)
(declare-fun T!val!11 () T)
(declare-fun T!val!3 () T)
(declare-fun T!val!14 () T)
(declare-fun T!val!0 () T)
(declare-fun T!val!4 () T)
(declare-fun T!val!2 () T)
(declare-fun T!val!13 () T)
(declare-fun T!val!15 () T)
(declare-fun T!val!9 () T)
(declare-fun T!val!7 () T)
(declare-fun T!val!1 () T)
;; cardinality constraint:
(forall ((x T))
        (or (= x T!val!12)
            (= x T!val!6)
            (= x T!val!8)
            (= x T!val!5)
            (= x T!val!10)
            (= x T!val!11)
            (= x T!val!3)
            (= x T!val!14)
            (= x T!val!0)
            (= x T!val!4)
            (= x T!val!2)
            (= x T!val!13)
            (= x T!val!15)
            (= x T!val!9)
            (= x T!val!7)
            (= x T!val!1)))
;; -----------
(define-fun AXIOM_VT_s_rat () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TLocation () Bool
  true)
(define-fun cs2_x () T
  T!val!11)
(define-fun AX_TH_TRating_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TMichelin_NoSubOf_TDistance () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TRating () Bool
  true)
(define-fun AX_TH_TEvent_SubOf_TEvent () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_UNIQUE_TSnackBar () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TLocation () Bool
  true)
(define-fun AXIOM_VT_r2_rest () Bool
  true)
(define-fun RULE_BOUND_isBetterRestaurant () Bool
  true)
(define-fun s_rest () T
  T!val!7)
(define-fun AX_TH_TRating_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TRestaurant_SubOf_TRestaurant () Bool
  true)
(define-fun AX_TH_UNIQUE_TSite () Bool
  true)
(define-fun AX_TH_UNIQUE_TMichelin () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TDistance () Bool
  true)
(define-fun cs1_y () T
  T!val!10)
(define-fun AX_TH_TMichelin_SubOf_TRating () Bool
  true)
(define-fun cs2_y () T
  T!val!12)
(define-fun AX_TH_TRating_NoSubOf_TPrice () Bool
  true)
(define-fun AX_TH_TPrice_SubOf_TPrice () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TLocation () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TSite () Bool
  true)
(define-fun AXIOM_SETOPS_OR () Bool
  true)
(define-fun s_rat () T
  T!val!8)
(define-fun RULE_RATINGs_AND_RESTAURANTS () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_UNIQUE_TEvent () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TPrice () Bool
  true)
(define-fun bound_isBetterRestaurant_other () T
  T!val!0)
(define-fun AX_TH_TPrice_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TRating () Bool
  true)
(define-fun AX_TH_TSnackBar_SubOf_TSite () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TEvent () Bool
  true)
(define-fun arr_TRating_supertypes () (Array Int Bool)
  (_ as-array k!917))
(define-fun AX_TH_TPrice_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TRestaurant () Bool
  true)
(define-fun AXIOM_PP_isRatingLess_IS_IRREFLEXIVE () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TPrice () Bool
  true)
(define-fun AXIOM_SETOPS_NOT () Bool
  true)
(define-fun bound_isRatingLess () T
  T!val!5)
(define-fun AXIOM_PP_hasPrice_IS_FUNCTIONAL () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TDistance () Bool
  true)
(define-fun AXIOM_SETOPS_AND () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TPrice () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TRestaurant () Bool
  true)
(define-fun AXIOM_VT_r1_e2 () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TSite () Bool
  true)
(define-fun AXIOM_PP_hasLocation_IS_FUNCTIONAL () Bool
  true)
(define-fun arr_TMichelin_supertypes () (Array Int Bool)
  (_ as-array k!918))
(define-fun AX_TH_TEvent_NoSubOf_TDistance () Bool
  true)
(define-fun r1_e2 () T
  T!val!2)
(define-fun AX_TH_UNIQUE_TPrice () Bool
  true)
(define-fun cs1_y!0 () T
  T!val!14)
(define-fun AX_TH_TMichelin_NoSubOf_TSite () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TPrice () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TDistance () Bool
  true)
(define-fun arr_TPrice_supertypes () (Array Int Bool)
  (_ as-array k!919))
(define-fun AXIOM_VT_r3_rat () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TDistance () Bool
  true)
(define-fun AXIOM_PP_startsAt_IS_FUNCTIONAL () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TRestaurant () Bool
  true)
(define-fun AXIOM_PP_distFrom_IS_FUNCTIONAL () Bool
  true)
(define-fun arr_TDate_supertypes () (Array Int Bool)
  (_ as-array k!923))
(define-fun CONSTRAINT_INSTANTIATION () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TLocation () Bool
  true)
(define-fun AXIOM_VT_bound_isBetterRestaurant () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TDistance () Bool
  true)
(define-fun AXIOM_PP_isRatingLess_IS_ANTISYMMETRIC () Bool
  true)
(define-fun r1_p1 () T
  T!val!3)
(define-fun AX_TH_TSnackBar_NoSubOf_TLocation () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TSite () Bool
  true)
(define-fun AXIOM_PP_distTo_IS_FUNCTIONAL () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TSite () Bool
  true)
(define-fun AX_TH_TLocation_SubOf_TLocation () Bool
  true)
(define-fun AXIOM_VT_bound_isRatingLess_other () Bool
  true)
(define-fun AXIOM_VT_r3_rest () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TMichelin_NoSubOf_TPrice () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TMichelin () Bool
  true)
(define-fun AXIOM_VT_bound_isRatingLess () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AXIOM_PP_isRatingLess_IS_TRICHOTOMOUS () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TDate () Bool
  true)
(define-fun AXIOM_PP_isBetterRestaurant_IS_ANTISYMMETRIC () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TRating () Bool
  true)
(define-fun AXIOM_VT_r1_p1 () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TEvent () Bool
  true)
(define-fun r1_e1 () T
  T!val!1)
(define-fun AX_TH_TMichelin_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TMichelin_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TDistance () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_UNIQUE_TRating () Bool
  true)
(define-fun AX_TH_TMichelin_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TRating_SubOf_TRating () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TLocation () Bool
  true)
(define-fun arr_TRestaurant_supertypes () (Array Int Bool)
  (_ as-array k!915))
(define-fun cs2_y!2 () T
  T!val!15)
(define-fun AX_TH_TMichelin_NoSubOf_TRestaurant () Bool
  true)
(define-fun AX_TH_UNIQUE_TDistance () Bool
  true)
(define-fun AX_TH_TSnackBar_SubOf_TRestaurant () Bool
  true)
(define-fun AX_TH_TSnackBar_NoSubOf_TPrice () Bool
  true)
(define-fun r2_rest () T
  T!val!4)
(define-fun AX_TH_TMichelin_NoSubOf_TLocation () Bool
  true)
(define-fun AX_TH_TMichelin_SubOf_TMichelin () Bool
  true)
(define-fun AXIOM_VT_r2_rat () Bool
  true)
(define-fun r2_rat () T
  T!val!5)
(define-fun AX_TH_TPrice_NoSubOf_TRating () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TPrice () Bool
  true)
(define-fun AXIOM_VT_bound_isBetterRestaurant_other () Bool
  true)
(define-fun AXIOM_VT_r1_e1 () Bool
  true)
(define-fun AX_TH_TDistance_SubOf_TDistance () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TDistance () Bool
  true)
(define-fun AXIOM_PP_isBetterRestaurant_IS_IRREFLEXIVE () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TSite () Bool
  true)
(define-fun arr_TSite_supertypes () (Array Int Bool)
  (_ as-array k!914))
(define-fun arr_TLocation_supertypes () (Array Int Bool)
  (_ as-array k!920))
(define-fun AX_TH_UNIQUE_TDate () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TSnackBar_SubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_UNIQUE_TLocation () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TRestaurant () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TRating () Bool
  true)
(define-fun AX_TH_TDate_NoSubOf_TMichelin () Bool
  true)
(define-fun AXIOM_VT_cs2_y () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TLocation () Bool
  true)
(define-fun AX_TH_UNIQUE_TRestaurant () Bool
  true)
(define-fun AX_TH_TSite_NoSubOf_TDate () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TEvent () Bool
  true)
(define-fun AX_TH_TRating_NoSubOf_TLocation () Bool
  true)
(define-fun AX_TH_TDate_SubOf_TDate () Bool
  true)
(define-fun arr_TSnackBar_supertypes () (Array Int Bool)
  (_ as-array k!916))
(define-fun AX_TH_TSite_SubOf_TSite () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TMichelin () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TRating () Bool
  true)
(define-fun arr_TEvent_supertypes () (Array Int Bool)
  (_ as-array k!922))
(define-fun r3_rest () T
  T!val!2)
(define-fun AX_TH_TDate_NoSubOf_TRestaurant () Bool
  true)
(define-fun AXIOM_PP_isBetterRestaurant_IS_TRANSITIVE () Bool
  true)
(define-fun cs1_x () T
  T!val!9)
(define-fun cs1_x!1 () T
  T!val!13)
(define-fun AXIOM_VT_cs1_x () Bool
  true)
(define-fun arr_TDistance_supertypes () (Array Int Bool)
  (_ as-array k!921))
(define-fun AX_TH_TSnackBar_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AX_TH_TRestaurant_NoSubOf_TPrice () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TMichelin () Bool
  true)
(define-fun AXIOM_VT_r1_p2 () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TDate () Bool
  true)
(define-fun AXIOM_VT_cs2_x () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TSite () Bool
  true)
(define-fun AX_TH_TDistance_NoSubOf_TRating () Bool
  true)
(define-fun AXIOM_PP_isBetterRestaurant_IS_TRICHOTOMOUS () Bool
  true)
(define-fun RULE_BOUND_isRatingLess () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TRestaurant () Bool
  true)
(define-fun bound_isRatingLess_other () T
  T!val!12)
(define-fun r1_p2 () T
  T!val!5)
(define-fun cs2_x!3 () T
  T!val!2)
(define-fun bound_isBetterRestaurant () T
  T!val!9)
(define-fun AX_TH_TDate_NoSubOf_TSnackBar () Bool
  true)
(define-fun AXIOM_PP_hasRating_IS_FUNCTIONAL () Bool
  true)
(define-fun AXIOM_VT_cs1_y () Bool
  true)
(define-fun AX_TH_TPrice_NoSubOf_TRestaurant () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TSetOfRestaurants () Bool
  true)
(define-fun AXIOM_PP_isRatingLess_IS_TRANSITIVE () Bool
  true)
(define-fun AXIOM_VT_s_rest () Bool
  true)
(define-fun r3_rat () T
  T!val!6)
(define-fun AX_TH_TRestaurant_SubOf_TSite () Bool
  true)
(define-fun AX_TH_TLocation_NoSubOf_TRating () Bool
  true)
(define-fun AX_TH_TMichelin_NoSubOf_TSnackBar () Bool
  true)
(define-fun AX_TH_TEvent_NoSubOf_TDate () Bool
  true)
(define-fun k!919 ((x!0 Int)) Bool
  (ite (= x!0 7) true
    false))
(define-fun setops_and ((x!0 Bool) (x!1 Bool)) Bool
  (not (or (not x!0) (not x!1))))
(define-fun IsOfTypeTSite ((x!0 T)) Bool
  false)
(define-fun k!918 ((x!0 Int)) Bool
  (ite (= x!0 5) true
  (ite (= x!0 6) true
    false)))
(define-fun IsOfTypeTDate ((x!0 T)) Bool
  false)
(define-fun Pred_hasLocation ((x!0 T) (x!1 T)) Bool
  false)
(define-fun k!917 ((x!0 Int)) Bool
  (ite (= x!0 5) true
    false))
(define-fun k!925 ((x!0 T)) T
  (ite (= x!0 T!val!11) T!val!11
  (ite (= x!0 T!val!2) T!val!2
  (ite (= x!0 T!val!5) T!val!5
  (ite (= x!0 T!val!4) T!val!4
  (ite (= x!0 T!val!8) T!val!8
  (ite (= x!0 T!val!7) T!val!7
  (ite (= x!0 T!val!0) T!val!0
  (ite (= x!0 T!val!9) T!val!9
  (ite (= x!0 T!val!1) T!val!1
  (ite (= x!0 T!val!3) T!val!3
  (ite (= x!0 T!val!13) T!val!13
  (ite (= x!0 T!val!6) T!val!6
  (ite (= x!0 T!val!14) T!val!14
  (ite (= x!0 T!val!10) T!val!10
  (ite (= x!0 T!val!15) T!val!15
    T!val!12))))))))))))))))
(define-fun k!928 ((x!0 T)) T
  (ite (= x!0 T!val!2) T!val!2
  (ite (= x!0 T!val!12) T!val!12
  (ite (= x!0 T!val!5) T!val!5
  (ite (= x!0 T!val!4) T!val!4
  (ite (= x!0 T!val!9) T!val!9
  (ite (= x!0 T!val!8) T!val!8
  (ite (= x!0 T!val!7) T!val!7
  (ite (= x!0 T!val!0) T!val!0
  (ite (= x!0 T!val!1) T!val!1
  (ite (= x!0 T!val!3) T!val!3
  (ite (= x!0 T!val!13) T!val!13
  (ite (= x!0 T!val!6) T!val!6
  (ite (= x!0 T!val!14) T!val!14
  (ite (= x!0 T!val!10) T!val!10
  (ite (= x!0 T!val!15) T!val!15
    T!val!11))))))))))))))))
(define-fun Pred_hasRating!937 ((x!0 T) (x!1 T)) Bool
  (ite (and (= x!0 T!val!2) (= x!1 T!val!15)) true
    false))
(define-fun Pred_hasRating ((x!0 T) (x!1 T)) Bool
  (Pred_hasRating!937 (k!928 x!0) (k!925 x!1)))
(define-fun IsOfTypeTMichelin ((x!0 T)) Bool
  false)
(define-fun IsOfTypeTRating!938 ((x!0 T)) Bool
  (ite (= x!0 T!val!3) true
  (ite (= x!0 T!val!5) true
  (ite (= x!0 T!val!6) true
  (ite (= x!0 T!val!8) true
  (ite (= x!0 T!val!10) true
  (ite (= x!0 T!val!12) true
    false)))))))
(define-fun k!916 ((x!0 Int)) Bool
  (ite (= x!0 3) true
  (ite (= x!0 4) true
  (ite (= x!0 2) true
    false))))
(define-fun k!915 ((x!0 Int)) Bool
  (ite (= x!0 3) true
  (ite (= x!0 2) true
    false)))
(define-fun IsOfTypeTSnackBar ((x!0 T)) Bool
  false)
(define-fun Pred_hasPrice ((x!0 T) (x!1 T)) Bool
  false)
(define-fun k!914 ((x!0 Int)) Bool
  (ite (= x!0 2) true
    false))
(define-fun IsOfTypeTPrice ((x!0 T)) Bool
  false)
(define-fun Pred_isGoodRestaurant ((x!0 T)) Bool
  true)
(define-fun setops_or ((x!0 Bool) (x!1 Bool)) Bool
  (or x!0 x!1))
(define-fun k!924 ((x!0 T)) T
  (ite (= x!0 T!val!12) T!val!12
  (ite (= x!0 T!val!5) T!val!5
  (ite (= x!0 T!val!4) T!val!4
  (ite (= x!0 T!val!9) T!val!9
  (ite (= x!0 T!val!2) T!val!2
  (ite (= x!0 T!val!7) T!val!7
  (ite (= x!0 T!val!0) T!val!0
  (ite (= x!0 T!val!8) T!val!8
  (ite (= x!0 T!val!1) T!val!1
  (ite (= x!0 T!val!3) T!val!3
  (ite (= x!0 T!val!6) T!val!6
  (ite (= x!0 T!val!10) T!val!10
    T!val!11)))))))))))))
(define-fun IsOfTypeTRating ((x!0 T)) Bool
  (IsOfTypeTRating!938 (k!924 x!0)))
(define-fun Pred_isBetterRestaurant!939 ((x!0 T) (x!1 T)) Bool
  (ite (and (= x!0 T!val!14) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!4) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!5)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!11)) true
    false)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
    ))))))))))))))))))))))))))))))))))))))))))))))))))
(define-fun IsOfTypeTDistance ((x!0 T)) Bool
  false)
(define-fun IsOfTypeTEvent ((x!0 T)) Bool
  false)
(define-fun Pred_isMinRating ((x!0 T)) Bool
  false)
(define-fun Pred_distFrom ((x!0 T) (x!1 T)) Bool
  false)
(define-fun IsOfTypeTRestaurant!940 ((x!0 T)) Bool
  (ite (= x!0 T!val!3) false
  (ite (= x!0 T!val!5) false
  (ite (= x!0 T!val!6) false
  (ite (= x!0 T!val!8) false
  (ite (= x!0 T!val!10) false
  (ite (= x!0 T!val!12) false
    true)))))))
(define-fun Pred_isRatingLess!941 ((x!0 T) (x!1 T)) Bool
  (ite (and (= x!0 T!val!11) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!13)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!0)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!12) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!6) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!11)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!7) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!5) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!15)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!14)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!12)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!10)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!1) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!0) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!13) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!15) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!8) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!3) (= x!1 T!val!2)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!4)) true
  (ite (and (= x!0 T!val!10) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!9) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!7)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!1)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!11) (= x!1 T!val!3)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!9)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!8)) true
  (ite (and (= x!0 T!val!14) (= x!1 T!val!6)) true
  (ite (and (= x!0 T!val!2) (= x!1 T!val!9)) true
    false))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
    ))))))))))))))))))))))))))))))))))))))))))))))))))
(define-fun Pred_distTo ((x!0 T) (x!1 T)) Bool
  false)
(define-fun Pred_startsAt ((x!0 T) (x!1 T)) Bool
  false)
(define-fun setops_not ((x!0 Bool)) Bool
  (not x!0))
(define-fun k!923 ((x!0 Int)) Bool
  (ite (= x!0 11) true
    false))
(define-fun IsOfTypeTRestaurant ((x!0 T)) Bool
  (IsOfTypeTRestaurant!940 (k!924 x!0)))
(define-fun k!922 ((x!0 Int)) Bool
  (ite (= x!0 10) true
    false))
(define-fun Pred_isBetterRestaurant ((x!0 T) (x!1 T)) Bool
  (Pred_isBetterRestaurant!939 (k!928 x!0) (k!928 x!1)))
(define-fun k!921 ((x!0 Int)) Bool
  (ite (= x!0 9) true
    false))
(define-fun IsOfTypeTLocation ((x!0 T)) Bool
  false)
(define-fun Pred_isRatingLess ((x!0 T) (x!1 T)) Bool
  (Pred_isRatingLess!941 (k!925 x!0) (k!925 x!1)))
(define-fun k!920 ((x!0 Int)) Bool
  (ite (= x!0 8) true
    false))
```

Bibliography

Good relations ontology project. `http://www.goodrelations-vocabulary.org`.

OpenGALEN ontology. `http://www.opengalen.org`.

Qallme tourism ontology. `http://qallme.fbk.eu`.

Schema.org e-commerce taxonomy. `http://www.schema.org`.

Jean-Raymond Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In Søren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods: 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, October 21–25, 1991 Proceedings*, pages 398–405, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6): 509–516, June 1978.

Anupriya Ankolekar, Massimo Paolucci, and Katia Sycara. Towards a formal verification of OWL-S process models. In Yolanda Gil, Enrico Motta, V.Richard Benjamins, and MarkA. Musen, editors, *The Semantic Web – ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2005.

Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 482–491, Piscataway, NJ, USA, 2013. IEEE Press.

K. Apt, F. de Boer, and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer, 2009.

Svetlana Arifulina, Matthias Becker, Marie Platenius, and Sven Walther. SeSAME: Modeling and analyzing high-quality service compositions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 839–842. ACM, 2014.

Svetlana Arifulina, Marie Christin Platenius, Matthias Becker, Gregor Engels, and Wilhelm Schäfer. An overview of service specification language and matching in on-the-fly computing. Technical Report tr-ri-15-347, Heinz Nixdorf Institute, University of Paderborn, July 2015.

Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, February 2009.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 135 – 179. Elsevier, 2008.

Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 21–43. Springer Berlin Heidelberg, 2009.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, pages 49–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007. Proceedings*, pages 378–394, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005. Proceedings*, pages 113–129, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In Sophia Drossopoulou, editor, *Programming Languages and Systems: 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 148–162, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS 2013*, pages 93–107, 2013.

Edmund Clarke, Daniel Kröning, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS 2004*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs: Workshop, Yorktown Heights, New York, May 1981*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.

David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, pages 108–128, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification*, pages 1–8, 2000.

Thomas H. Davenport. *Process innovation : reengineering work though information technology*. Boston, Mass. : Harvard Business School Press, [reprint] edition, 1998.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

Randall Davis, Howard Shrobe, and Peter Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 45–52, Nov 2009.

Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Al-log: Integrating datalog and description logics. *J. Intell. Inf. Syst.*, 10(3):227–252, 1998.

Marlon Dumas and Arthur H. M. ter Hofstede. Uml activity diagrams as a workflow specification language. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings*, pages 76–90, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

J. Erickson. Trust metrics. In *Collaborative Technologies and Systems, 2009. CTS '09. International Symposium on*, pages 93–97, May 2009.

Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. A formal semantics for the business process execution language for web services. In Savitri Bevinakoppa, Luís Ferreira Pires, and Slimane Hammoudi, editors, *Web Services and Model-Driven Enterprise Information Services, Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Services, WSMDEIS 2005, In conjunction with ICEIS 2005, Miami, USA, May 2005*, pages 122–133. INSTICC Press, 2005.

Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema. W3C recommendation, W3C, August 2007. http://www.w3.org/TR/2007/REC-sawsdl-20070828/.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM.

Jean-Christophe Filliâtre. One logic to use them all. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24: 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

T. Fleuren, J. Götze, and P. Müller. Workflow skeletons: A non-intrusive approach for facilitating scientific workflow modeling. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 459–466, Aug 2014.

Robert W. Floyd. Assigning meanings to programs. volume 19, pages 19–32. American Mathematical Society, 1967.

Enrico Franconi and Sergio Tessaris. Rules and queries with ontologies: A unified logical framework. In Hans Jürgen Ohlbach and Sebastian Schaffert, editors, *Principles and Practice of Semantic Web Reasoning*, volume 3208 of *Lecture Notes in Computer Science*, pages 50–60. Springer Berlin Heidelberg, 2004.

Martin Fränzle, Holger Hermanns, and Tino Teige. Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In Magnus Egerstedt and Bud Mishra, editors, *Hybrid Systems: Computation and Control*, number 4981 in Lecture Notes in Computer Science, pages 172–186. Springer Berlin Heidelberg, 2008.

Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, pages 277–300, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Werner Gabrisch and Wolf Zimmermann. A hoare-style verification calculus for control state ASMs. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 205–210, New York, NY, USA, 2012. ACM.

Juan Pablo Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. Dynamate: Dynamically inferring loop invariants for automatic full functional verification. In Eran Yahav, editor, *Hardware and Software: Verification and Testing: 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, pages 48–53, Cham, 2014. Springer International Publishing.

Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977*, Advances in Data Base Theory, New York, 1978. Plemum Press.

Stefano Gallotti, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *Quality of Software Architectures. Models and Architectures*, pages 119–134. Springer, 2008.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Manuel Gesell and Klaus Schneider. A Hoare calculus for the verification of synchronous languages. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, pages 37–48, New York, NY, USA, 2012. ACM.

Y. Gil, P. Groth, V. Ratnakar, and C. Fritz. Expressive reusable workflow templates. In *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, pages 344–351, Dec 2009.

Y. Gil, V. Ratnakar, J. Kim, P. Gonzalez-Calero, P. Groth, J. Moody, and E. Deelman. Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1):62–72, Jan 2011.

Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

Bernardo Cuenca Grau, Peter Patel-Schneider, and Boris Motik. OWL 2 web ontology language direct semantics (second edition). W3C recommendation, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.

David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer, 1981.

Object Management Group. UML specification. `http://www.omg.org/spec/UML/`, retrieved 2016, June 9.

Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.

Orna Grumberg and DavidE. Long. Model checking and modular verification. In JosC.M. Baeten and JanFrisco Groote, editors, *CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*, pages 250–265. Springer Berlin Heidelberg, 1991.

Nicola Guarino, Daniel Oberle, and Steffen Staab. What is an ontology? In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 1–17. Springer Berlin Heidelberg, 2009.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM.

Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 634–640, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Volker Haarslev, Roberto Sebastiani, and Michele Vescovi. Automated reasoning in ALCQ via SMT. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2011.

M. Happe, F. Meyer auf der Heide, P. Kling, M. Platzner, and C. Plessl. On-the-fly computing: A novel paradigm for individualized it services. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–10, June 2013.

Patrick Hayes and Peter Patel-Schneider. RDF 1.1 semantics. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

David Hemer and Peter A. Lindsay. An industrial-strength method for the construction of formally verified software. *Australian Software Engineering Conference*, page 27, 1996.

David Hemer and Peter A. Lindsay. Reuse of verified design templates through extended pattern matching. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 495–514. Springer, 1997.

Reiner Hähnle and Ina Schaefer. A Liskov principle for delta-oriented programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2012.

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

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

Michael Holzinger, editor. *Aristoteles: Organon.* CreateSpace Independent Publishing Platform, 2013.

Matthew Horridge and Peter Patel-Schneider. OWL 2 web ontology language manchester syntax (second edition). W3C note, W3C, December 2012. http://www.w3.org/TR/2012/NOTE-owl2-manchester-syntax-20121211/.

Matthew Horridge, Tania Tudorache, Csongor Nyulas, and Mark A. Musen. Webprotégé: a web-based development environment for OWL ontologies. In C. Maria Keet and Valentina A. M. Tamma, editors, *Proceedings of the 11th International Workshop on OWL: Experiences and Directions (OWLED 2014) co-located with 13th International Semantic Web Conference on (ISWC 2014), Riva del Garda, Italy, October 17-18, 2014.*, volume 1265 of *CEUR Workshop Proceedings*, pages 109–120. CEUR-WS.org, 2014.

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Marie-Christine Jakobs and Heike Wehrheim. Certification for configurable program analysis. In *Proceedings of the 21st International Symposium on Model Checking of Software (SPIN)*, SPIN 2014, pages 30–39. ACM, 2014.

Alexander Jungmann, Sonja Brangewitz, Ronald Petrlic, and Marie Christin Platenius. Towards a flexible and privacy-preserving reputation system for markets of composed services. In *SERVICE COMPUTATION 2014, The Sixth International Conferences on Advanced Service Computing*, pages 49–57, 2014.

Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6 (2):133–151, 1976.

Holger Knublauch, Matthew Horridge, Mark A. Musen, Alan L. Rector, Robert Stevens, Nick Drummond, Phillip W. Lord, Natalya Fridman Noy, Julian Seidenberg, and Hai Wang. The protege OWL experience. In Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, and Peter F. Patel-Schneider, editors, *Proceedings of the OWLED*05 Workshop on OWL: Experiences and Directions, Galway, Ireland, November 11-12, 2005*, volume 188 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

Andreas Krakau. Entwicklung eines Konzeptes zur Kodierung eines objektorientierten Typsystems in SMT, 2014. Bachelor's thesis, University of Paderborn.

Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the SMT-Lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22*, 2009.

Sri Krishna Kumar and J. A. Harding. Ontology mapping using description logic and bridging axioms. *Computers in Industry*, 64(1):19–28, 2013.

Marta Kwiatkowska and David Parker. Advances in probabilistic model checking. *Proc. 2011 Marktoberdorf Summer School: Tools for Analysis and Verification of Software Safety and Security IOS Press*, 2012.

Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification*, pages 585–591. Springer, 2011.

Shuvendu Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. *ACM SIGPLAN Notices*, 43(1):171–182, 2008.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral*

Specifications of Businesses and Systems, pages 175–188, Boston, MA, 1999. Springer US.

C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, July 1959.

K. Leino and Rustan M. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Alon Y. Levy and Marie-Christine Rousset. The limits on combining recursive horn rules with description logics. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.*, pages 577–584. AAAI Press / The MIT Press, 1996.

Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

Michael L Littman, Stephen M Majercik, and Toniann Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2001.

C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1):89 – 106, 2004.

Félix Gómez Mármol and Marcus Quintino Kuhnen. Reputation-based web service orchestration in cloud computing: A survey. *Concurrency and Computation: Practice and Experience*, 27(9):2390–2412, 2015.

David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The OWL-S approach. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer Berlin Heidelberg, 2005.

Bertrand Meyer. A basis for the constructive approach to programming. In *IFIP Congress*, pages 293–298, 1980.

Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, Oct 1992.

Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

Aleksandar Milicevic and Hillel Kugler. Model checking using SMT and theory of lists. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 282–297, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Felix Mohr and Sven Walther. Template-based generation of semantic services. In *Proceedings of the 14th International Conference on Software Reuse (ICSR)*, LNCS, pages 188–203. Springer, 2014.

F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, April 1984.

Boris Motik, Ulrike Sattler, and Rudi Studer. Query answering for OWL-DL with rules. *J. Web Sem.*, 3(1):41–60, 2005.

Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, and Ulrike Sattler. Representing ontologies using description logics, description graphs, and rules. *Artificial Intelligence*, 173(14):1275 – 1309, 2009.

George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 106–119, New York, NY, USA, 1997. ACM.

George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

Natalya F. Noy. Ontology mapping. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 573–590. Springer, 2009.

OASIS. Web services business process execution language. `https://www.oasis-open.org/committees/wsbpel`, retrieved July 21, 2017.

Chun Ouyang, Eric Verbeek, Wil M.P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H.M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2):162 – 198, 2007.

Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

Bijan Parsia, Evren Sirin, Bernardo Grau, Edna Ruckhaus, and Daniel Hewlett. Cautiously approaching SWRL. 2005. Preprint submitted to Elsevier Science.

Peter Patel-Schneider, Bijan Parsia, and Boris Motik. OWL 2 web ontology language structural specification and functional-style syntax (second edition). W3C recommendation, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

Gordon D. Plotkin. *A structural approach to operational semantics.* DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004 Venice, Italy, January 11-13, 2004 Proceedings*, pages 239–251, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Riccardo Rosati. Towards expressive KR systems integrating datalog and description logics: preliminary report. In Patrick Lambrix, Alexander Borgida, Maurizio Lenzerini, Ralf Möller, and Peter F. Patel-Schneider, editors, *Proceedings of the 1999 International Workshop on Description Logics (DL'99), Linköping, Sweden, July 30 - August 1, 1999*, volume 22 of *CEUR Workshop Proceedings*. CEUR-WS.org, 1999.

Riccardo Rosati. On the decidability and complexity of integrating ontologies and rules. *J. Web Sem.*, 3(1):61–73, 2005.

Maryam Sanati. Formal semantics of probalistic SMT solving in verification of service compositions, 2014. Master's thesis, University of Paderborn.

Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1 – 26, 1991.

Michael Schneider. OWL 2 web ontology language RDF-based semantics (second edition). W3C recommendation, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/.

Dana S. Scott. Mathematical concepts in programming language semantics. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972*, pages 225–234, 1972.

Roberto Sebastiani and Michele Vescovi. Encoding the satisfiability of modal and description logics into SAT: the case study of K(m)/ALC. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pages 130–135, 2006.

Roberto Sebastiani and Michele Vescovi. Automated reasoning in modal and description logics via SAT encoding: the case study of K(m)/ALC-satisfiability. *J. Artif. Intell. Res. (JAIR)*, 35:343–389, 2009.

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird,* *UT, USA, July 14-20, 2011. Proceedings*, pages 703–719, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Evren Sirin, Bijan Parsia, and James Hendler. Template-based composition of semantic web services. 2005.

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53, 2007. Software Engineering and the Semantic Web.

Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations*. MetaLibri, 2007.

Siavash Soleimanifard and Dilian Gurov. Algorithmic verification of procedural programs in the presence of code variability. In Ivan Lanese and Eric Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *Lecture Notes in Computer Science*, pages 327–345. Springer International Publishing, 2015.

Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 223–234, New York, NY, USA, 2009. ACM.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

Clemens A. Szyperski. *Component software - beyond object-oriented programming*. Addison-Wesley-Longman, 1998.

Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Logics in Artificial Intelligence: 8th European Conference, JELIA 2002 Cosenza, Italy, September 23–26, 2002 Proceedings*, pages 308–319, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, pages 292–297, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

W. M. P. van der Aalst and A. H. M. ter Hofstede. Workflow patterns put into context. *Software & Systems Modeling*, 11(3):319–323, 2012.

Wil van der Aalst and Kees Max van Hee. *Workflow management : models, methods, and systems.* Cooperative information systems. Cambridge, Mass. [u.a.] : MIT Press, 2002.

Wil van der Aalst, Michael Beisiegel, Kees van Hee, Dieter König, and Christian Stahl. A SOA-based architecture framework. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

Michele Vescovi. Exploiting SAT and SMT techniques for automated reasoning and ontology manipulation in description logics, 2011. PhD thesis, University of Trento.

W3C. W3C Data Activities, a. `https://www.w3.org/2013/data/`.

W3C. W3C Semantic Web Activities, b. `https://www.w3.org/2001/sw/`.

Sven Walther and Heike Wehrheim. Knowledge-based verification of service compositions – an SMT approach. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 24–32. IEEE, 2013.

Sven Walther and Heike Wehrheim. Verified service compositions by template-based construction. In Ivan Lanese and Eric Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *Lecture Notes in Computer Science*, pages 31–48. Springer International Publishing, 2015.

Sven Walther and Heike Wehrheim. On-the-fly construction of provably correct service compositions – templates and proofs. *Science of Computer Programming*, 127:2 – 23, 2016. Special issue of the 11th International Symposium on Formal Aspects of Component Software.

Sanjiva Weerawarana, Arthur Ryman, Jean-Jacques Moreau, and Roberto Chinnici. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626.

Thomas Weise, M. Brian Blake, and Steffen Bleul. Semantic web service composition: The web service challenge perspective. In Athman Bouguettaya, Z. Quan Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 161–187, New York, NY, 2014. Springer New York.

D. Wonisch, A. Schremmer, and H. Wehrheim. Programs from proofs – a PCC alternative. In H. Veith N. Sharygina, editor, *Computer Aided Verification*, volume 8044, pages 912–927. Springer Berlin/Heidelberg, 2013a.

D. Wonisch, A. Schremmer, and H. Wehrheim. Programs from proofs – approach and applications. In N.C. Ehmke W. Hasselbring, editor, *Proceedings of the Software Engineering Conference (SE)*, volume 227, pages 67–68, 2014.

Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Zero overhead runtime monitoring. In RobertM. Hierons, MercedesG. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods*, volume 8137 of *LNCS*, pages 244–258. Springer Berlin Heidelberg, 2013b.

Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

Zhengping Wu and A.C. Weaver. Application of fuzzy logic in federated trust management for pervasive computing. In *COMPSAC*, volume 2, pages 215–222, 2006.