Verified Service Compositions by Template-Based Construction*

Sven Walther and Heike Wehrheim

Department of Computer Science
University of Paderborn, Germany
{swalther,wehrheim}@mail.upb.de

Abstract. Today, service compositions often need to be assembled or changed on-the-fly, which leaves only little time for quality assurance. Moreover, quality assurance is complicated by service providers only giving information on their services in terms of *domain specific* concepts with only limited semantic meaning.
In this paper, we propose a method to construct service compositions based on *pre-verified templates*. Templates, given as workflow descriptions, are typed over a (domain-independent) template ontology defining concepts and predicates. Templates are proven correct using an *abstract semantics*, leaving the specific meaning of ontology concepts open, however, only up to given *ontology rules*. Construction of service compositions amounts to instantiation of templates with domain-specific services. Correctness of an instantiation can then simply be checked by verifying that the domain ontology (a) adheres to the rules of the template ontology, and (b) fulfills the constraints of the employed template.

1 Introduction

Concepts like component-based software engineering (CBSE) or service-oriented architectures (SOA) ease the construction of software by combining off-the-shelf components or services to compositions. Today, such compositions often need to be assembled or changed *on-the-fly*, thereby imposing strong timing constraints on quality assurance. "Quality" of service compositions might refer to either non-functional properties (like performance [7]), or functional requirements like adherence to protocols (e.g., [8]), to given pre- and postconditions [21], or to properties specified with temporal logic [25]. Quality assurance methods typically translate the composition (e.g., an architecture model, or a workflow description) into an analysis model, which captures the semantics of the composition and allows – at the best – for a fully automatic quality analysis. Both the transformation into the analysis model and the analysis itself are time-costly and thus difficult to apply in an on-the-fly composition scenario.

In this paper, we propose a technique for service composition and analysis based on *templates*. Templates can capture known compositional patterns, and

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

thus allow for the generally proven principle of pattern usage in software engineering [12]. In this paper, templates are workflow descriptions with *service placeholders*, which are replaced with concrete services by instantiations. Our templates are already *verified*, i.e., all template instances will be *correct by construction*. Every template specification contains pre- and postconditions (with associated meaning "if precondition fulfilled then postcondition guaranteed"), and a correct template provably adheres to this specification. This poses a non-trivial task on verification: Since templates should be usable in a wide range of contexts and the instantiations of service placeholders are unknown at template design time, we cannot give a fixed semantics to templates. Rather, the template semantics needs to be *parameterized* in usage context and service instantiation. A template is only correct if it is correct for all (allowed) usage contexts.

Technically, we capture the usage contexts by *ontologies*, and the interpretation of concepts and predicates occurring therein by *logical structures*. A *template ontology* fixes the concepts and predicates of a template. Furthermore, a template specification contains *constraints* fixing additional conditions on instantiations. These constraints allow us to verify the correctness of the template despite unknown usage and unknown fixed semantics. A template instantiation replaces the template ontology with a homomorphous *domain ontology*, and the service placeholders with concrete services of this domain. Verification of the instantiation then amounts to checking whether the (instantiated) template constraints are valid within the domain ontology, and thus can be carried out on-the-fly.

Section 2 describes ontologies and logical structures. Section 3 continues with the syntax of templates, and Section 4 proceeds with their semantics and correctness. Section 5 explains instantiation and presents the central result of our approach: instantiation of correct templates yields correct service compositions, if constraints are respected. Section 6 discusses related work, and Section 7 concludes.

2 Foundations

We assume service compositions to be assembled of services which are specified by a signature and pre- and postconditions. Languages to describe signatures with pre- and postconditions are already in use (e.g., OWL-S [21]). Such service descriptions rely on domain specific concepts. *Ontologies* formally specify a conceptualization of domain knowledge [13]; the semantics can be defined, e.g., by *description logics* [5]. For this paper, we retain a high-level view of ontologies, and focus on *concepts*, *roles* (relating concepts), and *rules* (formalizing additional knowledge).

Definition 1. *Let C be a finite set of* concept *symbols, and P a finite set of* role *(or* predicate*) symbols, where every $p \in P$ denotes a unary or higher order relation on concepts. Let R be a set of rules of the form $b_0 \wedge \cdots \wedge b_n \to h_0 \wedge \cdots \wedge h_m$, where b_i, h_i are negated or non-negated predicates denoting concepts of C, or roles from P. Then $K = (C, P, R)$ denotes a* rule-enhanced ontology*.*

We assume every variable in the rules to be implicitly all-quantified, and rules to be *consistent*, i.e., not to contain contradictions. For details about ontologies and rules, we refer to [14,11]. In classical ontologies, predicates are always binary; however, roles (predicates) relating to boolean types can be expressed unary, and n-ary predicates with $n > 2$ can be translated to binary predicates by introducing supplementary concepts. To avoid technicalities, we allow for a more general notion of predicates here.

Example 1 introduces a template ontology. It will be used later in a "filter" template, which extracts "good" elements from a set using a filter predicate.

Example 1. The template ontology $K_T = (C_T, P_T, R_T)$ has two concepts, two predicates, and no rules:

$$C_T = \{\mathit{Elem}, \mathit{Value}\}, P_T = \{\mathit{fp} : \mathit{Value} \to \mathbf{bool}, g : \mathit{Elem} \to \mathbf{bool}\}, R_T = \{\} .$$

Signatures and pre- and postconditions are specified using concepts and roles, viz. predicates, of ontologies. To allow standard types like integers, we inductively define the set $\mathcal{T}_K$ of *types over an ontology* $K = (C_K, P_K, R_K)$ by these three rules: (i) $c \in C_K \to c \in \mathcal{T}_K$, (ii) $\mathbf{int} \in \mathcal{T}_K$, $\mathbf{bool} \in \mathcal{T}_K$, and (iii) $T \in \mathcal{T}_K \to \mathbf{set}\ T \in \mathcal{T}_K$. We furthermore assume that all predicate symbols of ontologies are typed as well[1], thus, e.g., a binary predicate p relating concepts T_1 and T_2 has type $T_1 \times T_2 \to \mathbf{bool}$. We therefore implicitly extend purely domain specific ontologies by standard types.

Types are used to fix the types of inputs and outputs of services; the predicates can occur in pre- and postconditions. We assume that we have – in addition to the predicates of an ontology K – standard predicate symbols, operations and constants on integers, booleans and sets available, e.g., $\mathit{true}, <, >, \leq, =, \in, \cup, \cap, \emptyset, \ldots$ These make up a set $\mathcal{P}_K$ (for predicate symbols) and a set $\mathcal{F}_K$ (for function symbols). We assume that $P_K \subset \mathcal{P}_K$. Note that the ontology itself does not define any function symbols. From $\mathcal{P}_K$ and $\mathcal{F}_K$, we construct first-order logic formulae in the usual way. To only get type-correct formulae, we assume a set of typed variables Var, i.e., given an ontology K we assume a typing function $\mathit{type} : \mathit{Var} \to \mathcal{T}_K$.

We assume typed *terms* based on function symbols $\mathcal{F}_K$ and typed variables to be defined in the usual way. Note that the set of terms over different ontologies might only use different variables, but always use the same (standard) function symbols; also constants like true or 1 are nullary function symbols. Using typed terms, we define the set of first-order logic formulae over K.

Definition 2. *Let K be an ontology with types $\mathcal{T}_K$, predicate symbols $\mathcal{P}_K$, and function symbols $\mathcal{F}_K$. The* set of first order formulae over K, Φ_K, *is inductively defined as follows:*

- *if $p \in \mathcal{P}_K$ is a predicate symbol of arity k and type $T_1 \times \ldots \times T_k \to$ **bool** and $e_1, \ldots e_k$ are terms of type $T_1, \ldots, T_k$, respectively, then $p(e_1, \ldots, e_k) \in \Phi_K$,*

[1] In expressive ontology languages and description logics, it is possible to express notions similar to sub-classing; as we restrict ourselves to a simple version of ontologies, we can assume our roles to be typed even with this simple type system.

– *if* $\varphi_1, \varphi_2 \in \Phi_K$ *then* $\neg\varphi_1 \in \Phi_K$ *and* $\varphi_1 \vee \varphi_2 \in \Phi_K$,
– *if* $\varphi \in \Phi_K$ *then* $\forall x : \varphi \in \Phi_K$ *and* $\exists x : \varphi \in \Phi_K$.

As usual, we write $\mathit{free}(F)$ to denote the free and $\mathit{bound}(F)$ for the bound variables of a formula F.

The meaning of first-order logic is usually defined with respect to a *logical structure*. A logical structure fixes the universe out of which elements of the types are taken as well as an interpretation of the predicate and function symbols.

Definition 3. *Let* K *be an ontology with types* $\mathcal{T}_K$*, predicate symbols* $\mathcal{P}_K$*, and function symbols* $\mathcal{F}_K$*. A* logical structure *over* K*,* $\mathcal{S}_\mathcal{K} = (\mathcal{U}, \mathcal{I})$*, consists of*

– $\mathcal{U} = \bigcup_{T \in \mathcal{T}_K} \mathcal{U}_T$ *the* universe *of values split up for the different types, and*
– $\mathcal{I}$ *an* interpretation *of the predicate and function symbols, i.e., for every* $p \in \mathcal{P}_K$ *of type* $T_1 \times \ldots \times T_K \to$ ***bool*** *and every* $f \in \mathcal{F}_K$ *of type* $T_1 \times \ldots \times T_K \to T$ *we have a predicate* $\mathcal{I}(p) : \mathcal{U}_{T_1} \times \ldots \times \mathcal{U}_{T_k} \to \mathcal{U}_{\mathbf{bool}}$ *and a function* $\mathcal{I}(f) : \mathcal{U}_{T_1} \times \ldots \times \mathcal{U}_{T_k} \to \mathcal{U}_T$*, respectively.*

We assume standard domains and interpretations for integers, sets, and boolean, e.g., $\mathcal{U}_{\mathbf{bool}} = \{true, false\}$. Therefore all logical structures of an ontology agree on standard types and their operations, but may differ on domain specific parts.

To define a semantics for formulae with free variables, we need a valuation of variables. We let $\sigma : V \to \mathcal{U}$ be a valuation of $V \subseteq \mathit{Var}$ with (type-correct) values from $\mathcal{U}$. We write $\sigma \models_\mathcal{S} F$ for a structure $\mathcal{S}$ and a formula F if $\mathcal{S}$ together with σ is a model for F (viz. F holds true in $\mathcal{S}$ and σ); refer to, e.g., [1] for a formal definition. If the formula contains no free variables, we can elide σ and just write $\models_\mathcal{S} F$, or $\mathcal{S} \models F$. Note that an ontology usually does not fix a structure because it neither gives a universe nor an interpretation for its predicates. It does, however, define constraints on valid interpretations by the rules R.

Definition 4. *A structure* $\mathcal{S}$ *over an ontology* $K = (C, P, R)$ satisfies the rules R *of the ontology,* $\mathcal{S} \models R$*, if it satisfies every rule in* R*, i.e.,* $\forall r \in R : \mathcal{S} \models r$*.*

Note that rules do not contain free variables, and therefore no σ is needed here.

3 Services and Templates

The ontology and logical formulae over the ontology are basic building blocks for services and compositions. A *service* in our notation is an entity which generates outputs for given inputs. The signature fixes the types of inputs and outputs.

Definition 5. *A* service signature *over an ontology* K *specifies the* name *of the service as well as the type, order, and number of inputs* $T_1 \times \cdots \times T_j$ *and the type, order, and number of outputs* $T_{j+1} \times \cdots \times T_k$*, where each* $T_i \in \mathcal{T}_K$*.*

Additionally, a service is specified by its *pre-* and *postcondition* (also: effect). Both of these are given as first-order formulae. They are formulated over a set of input and output variables.[2]

[2] In combination also known as IOPE (Input/Output/Precondition/Effect) in SOA.

Definition 6. *A* service description *of a service Svc over an ontology K consists of service signature, lists of input variables I and output variables O, a precondition pre_{Svc} and a postcondition $post_{Svc}$, both elements of Φ_K. Variables are typed according to the signature of Svc, that is, for $Svc : T_1 \times \cdots \times T_j \to T_{j+1} \times \cdots \times T_k$:*

- $I = (i_1, \ldots, i_j)$ *with* $type(i_l) = T_l$ *for all* $0 < l \leq j$, *and*
- $O = (i_{j+1}, \ldots, i_k)$ *with* $type(i_l) = T_l$ *for all* $j < l \leq k$.

The precondition describes only inputs, the postcondition inputs and outputs: $free(pre_{Svc}) \subseteq I$ *and* $free(post_{Svc}) \subseteq I \cup O$.

The set of service descriptions over an ontology K is denoted $\mathcal{SVC}_K$. Services are composed using a *workflow* describing the order of execution of the services. Workflows comprise control flow (using control structures) and data flow (using variables) between services. While different notations are in practical use (e.g., WS-BPEL [23]), we use a simple programming language style notation here.

Definition 7. *Let K be an ontology. The syntax of a* workflow W over K *can be described by the following rules:*

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

*with variables u, a, A; expression t of type $type(u)$; A of type **set** T; a of type $T \in \mathcal{T}_K$; $B \in \Phi_K$; and Svc a service call with service description Svc, with inputs $i_1, \ldots, i_j$, and outputs $u_{j+1}, \ldots, u_k$, with type and order fixed by the signature.*

Here, we augment usual imperative programming elements with an iteration construct for set types and with service calls. Workflows are build over arbitrary sets of services, defined on the same ontology. For template workflows, we do not use concrete services but service *placeholders*. Formally, a service placeholder has a signature like a service, but instead of formulae for pre- and postconditions we just write pre^{sp}_{Svc} and $post^{sp}_{Svc}$. We write $\mathcal{SP}_K$ to denote the set of service placeholders over K.

Example 2. The template in Fig. 1 accepts one input and produces one output, both of a set type with element type *Elem*. It uses one service placeholder, V, and the predicates *fp* and g from Example 1. Its workflow initializes the output variable A', and then iterates over the input set A. Every element is given to the service placeholder V, and then filtered by applying *fp* to the result of the service call. If filtering succeeds, the element is put in the output set A'.

Like services, templates have pre- and postconditions: they define the correctness properties which we intend to achieve with the template, and allow us to treat instantiated templates as any other services. The last part we find in a template are *constraints*. They define conditions on instantiations: if a template instantiation cannot guarantee these constraints, the postcondition of the template might not be achieved, i.e., the template concretion might not be correct. We will make this more precise in Section 5.

Name : FILTER
Inputs : A with $type(A) = \textbf{set } Elem$
Outputs : A' with $type(A') = \textbf{set } Elem$
Services : $V : Elem \rightarrow Value$
Precondition : $\{\forall a \in A : pre_V^{sp}(a)\}$
Postcondition: $\{A' = \{a \in A \mid g(a)\}\}$
Constraints : $\{\forall x, y : post_V^{sp}(x, y) \wedge fp(y) \Rightarrow g(x),$
$\forall x, y : post_V^{sp}(x, y) \wedge \neg fp(y) \Rightarrow \neg g(x)\}$

```
1 A' := ∅ ;
2 foreach a ∈ A do
3 |  (y) := V(a) ;
4 |  if fp(y) then  A' := A' ∪ {a} fi ;
5 od
```

Fig. 1. Template to filter a list, using a filter *fp* and a validation service V

Definition 8. *A* workflow template *WT over an ontology K consists of*

- *a name N,*
- *a list of typed input variables I and typed output variables O,*
- *a set of services placeholders $\mathcal{SP}_K$,*
- *a precondition $pre \in \Phi_K$ and a postcondition $post \in \Phi_K$,*
- *a set of constraint rules C as in Def. 1, and*
- *a workflow description W.*

In short: $WT = (N, I, O, \mathcal{SP}_K, pre, post, C, W)$.

Later, we see how templates get instantiated. To this end, we need concrete, existing services described by a *domain ontology*, to replace the service placeholders. However, our ultimate aim is to show correctness on the level of templates, and inherit their correctness onto instantiations. Thus, we will now define the semantics of templates and, with this help, their correctness.

4 Semantics of Templates

The key principle of our approach is to take correct templates, instantiate them, and afterwards be able to check correctness of instances by checking simple side-conditions. We start with the meaning of "correctness of templates". Figure 1 shows a template with pre- and postconditions and constraints. Basically, these state the property which the template should guarantee: if the precondition holds and the constraints are fulfilled, then the postcondition is achieved. All these parts contain undefined symbols: neither do we know the pre- and postconditions of the employed services (they are placeholders), nor the meaning of the predicates of the template ontology. The definition of a semantics of templates and their correctness therefore necessarily has to be *abstract*, i.e., defined modulo a concrete meaning. This meaning can only be fixed once we

have a logical structure. This is, however, not given by an ontology; thus the logical structure is a parameter for our semantics. The second parameter to the semantics is the concretion of service placeholders with actual services, given by a mapping $\pi : \mathcal{SP}_K \to \mathcal{SVC}_K$.

Definition 9. *Let K be an ontology, $\mathcal{SP}_K$ a set of service placeholders over K, and $\mathcal{SVC}_K$ be a set of service descriptions over K. Then $\pi : \mathcal{SP}_K \to \mathcal{SVC}_K$ is a* concretion of service placeholders*, if it respects signatures, i.e.*

$$\begin{aligned} &\textit{if } \pi(sp) = svc \textit{ and } sp : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_N \ , \\ &\qquad\qquad \textit{then } svc : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_N \ . \end{aligned}$$

We lift the definition of π to replace pre- and postconditions of service placeholders with their counterparts of the corresponding service, such that $\pi(pre^{sp}) \in \Phi_K$ with $free(\pi(pre^{sp})) = I_{svc}$, and $\pi(post^{sp}) \in \Phi_K$ with $free(\pi(post^{sp})) = I_{svc} \cup O_{svc}$. We use π to replace placeholders in any formula.

We define an operational semantics for workflows, much alike [1], however, always parameterized with a structure $\mathcal{S}$ and a concretion π. The semantics is defined by a transition relation between *configurations*. A configuration consists of a workflow to be executed and a state. We introduce a failure state `fail` which is entered once a service is called outside its preconditions. The workflow stops in `fail`, thus we define a *blocking semantics* for service calls here.

Definition 10. *Let $\mathcal{S} = (\mathcal{U}, \mathcal{I})$ be a structure of an ontology K, and Var a set of variables. A* state *σ over $\mathcal{S}$ is a type-correct mapping from Var to $\mathcal{U}$. The set of all states over $\mathcal{S}$ is denoted $\Sigma_\mathcal{S}$. We let $\Sigma_\mathcal{S}^{fail} = \Sigma_\mathcal{S} \cup \{\mathtt{fail}\}$.*

For a formula $F \in \Phi_K$ we define the set of states satisfying *F with respect to a structure $\mathcal{S}$ as $[\![F]\!]_\mathcal{S} = \{\sigma \in \Sigma_\mathcal{S} \mid \sigma \models_\mathcal{S} F\}$.*

A configuration *$\langle W, \tau \rangle$ has a workflow W over K and a state $\tau \in \Sigma_\mathcal{S}^{fail}$.*

We use E to stand for the empty workflow. Later, the semantics of workflows $[\![\cdot]\!]$ will map initial to final configurations. For this, we first define transitions between configurations by means of the set of axioms and rules given in Fig. 2. The main deviation from the standard semantics given in [1] is that we take two parameters into account: evaluation of conditions is parameterized in the interpretation of predicates as given in the structure $\mathcal{S}$, and influenced by the concretion of placeholders, π. We also add rules for the **foreach** statement and for service calls. Note that both introduce nondeterminism into the transition system: **foreach** iterates over the set of elements in an arbitrary order, and service calls can have more than one successor state.

Consider, e.g., rules (a) and (b) in Fig. 2: if a conditional statement is to be executed in state σ, then W_1 is selected as the next statement if and only if the condition B is true in the given structure $\mathcal{S}$ (with placeholders replaced); otherwise, W_2 is the next statement. In both cases, the state remains the same, as these rules only deal with the selection of the next workflow statement. Note that the states σ in the rules exclude the failure state, i.e., configurations $\langle W, \mathtt{fail} \rangle$ have no outgoing transitions.

$$\langle skip, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle$$

$$\langle u := t, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma[u := \sigma(t)] \rangle$$

$$\frac{\langle W_1, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W_2, \tau \rangle}{\langle W_1; W, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W_2; W, \tau \rangle}$$

$$\langle \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W_1, \sigma \rangle \qquad \text{if } \sigma \models_{\mathcal{S}} \pi(B) \qquad (a)$$

$$\langle \textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W_2, \sigma \rangle \qquad \text{if } \sigma \models_{\mathcal{S}} \neg\pi(B) \qquad (b)$$

$$\langle \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W; \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma \rangle$$
$$\text{if } \sigma \models \pi(B)$$

$$\langle \textbf{while } B \textbf{ do } W \textbf{ od}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle \qquad \text{if } \sigma \models \neg\pi(B)$$

$$\langle \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle E, \sigma \rangle \qquad \text{if } \sigma(A) = \emptyset$$

$$\langle \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle W; \textbf{foreach } a \in A \textbf{ do } W \textbf{ od}, \sigma' \rangle$$
$$\text{if } \sigma(A) \neq \emptyset \wedge \sigma'(a) = v \wedge$$
$$v \in \sigma(A) \wedge \sigma'(A) = \sigma(A) \setminus \{v\}$$

$$\langle (u_{j+1}, \ldots, u_k) := Svc(i_1, \ldots, i_j), \sigma \rangle \to^{\pi}_{\mathcal{S}}$$
$$\big\{ \langle E, \sigma[u_{j+1} := v_{j+1}, \ldots, u_k := v_k] \rangle \mid \pi(post^{sp}_{Svc}(\sigma(i_1), \ldots, \sigma(i_j), v_{j+1} \ldots, v_k)) \big\}$$
$$\text{if } \sigma \models_{\mathcal{S}} \pi(pre^{sp}_{Svc}(\sigma(i_1), \ldots, \sigma(i_j)))$$

$$\langle (u_{j+1}, \ldots, u_k) := Svc(i_1, \ldots, i_j), \sigma \rangle \to^{\pi}_{\mathcal{S}} \langle (u_{j+1}, \ldots, u_k) := Svc(i_1, \ldots, i_j), \texttt{fail} \rangle$$
$$\text{if } \sigma \models_{\mathcal{S}} \neg\pi(pre^{sp}_{Svc}(\sigma(i_1), \ldots, \sigma(i_j)))$$

Fig. 2. Transition axioms and rules based on [1], with additional rules for service calls and **foreach** constructs

The transition rules are used to derive the semantics of workflows. In this paper, we only define a partial correctness semantics, i.e., we do not specifically care about termination. Transitions lead to *transition sequences*, where a non-extensible transition sequence of a workflow W starting in σ is a *computation* of W. If it is finite and ends in $\langle E, \tau \rangle$ or $\langle W', \texttt{fail} \rangle$, then it *terminates.* We use the transitive, reflexive closure $\to^*$ of $\to$ to describe the effect of finite transition sequences. The semantics of partial correctness is again parameterized with a logical structure and a concretion mapping for service placeholders.

Definition 11. *Let $\mathcal{S}$ be a logical structure and π a concretion mapping. The* partial correctness semantics *of a workflow W with respect to $\mathcal{S}$ and π maps an initial state to a set of possible final states*

$$[\![W]\!]^{\pi}_{\mathcal{S}} : \Sigma_{\mathcal{S}} \to 2^{\Sigma_{\mathcal{S}} \cup \{\mathtt{fail}\}}, \ \textit{with } [\![W]\!]^{\pi}_{\mathcal{S}}(\sigma) = \{\tau \mid \langle W, \sigma \rangle {\to^{\pi}_{\mathcal{S}}}^{*} \langle W', \tau \rangle\}$$

where $W' = E$ or $\tau = \mathtt{fail}$.

We define a workflow template to be *correct*, if all computations starting in a state which satisfies the precondition, end in a state which fulfills the postcondition. Since services are only placeholders, correctness can only be stated when

the template works correctly for arbitrary concretions, as long as they obey the concretized constraints of the template. So far, we only operate on the template ontology, and thus arbitrary concretion means inserting arbitrary formulae for pre- and postconditions of placeholders. Therefore, our concretion mapping $\pi : \mathcal{SP}_K \to \mathcal{SVC}_K$ maps placeholders to arbitrary service descriptions over K.

Definition 12. *Let $WT = (N, I, O, \mathcal{SP}_K, pre, post, C, W)$ be a workflow template, and $K = (C_K, P_K, R_K)$ the corresponding ontology. We say WT is* correct *if the following holds:*

$$\forall \text{ logical structures } \mathcal{S} \text{ over } K, \forall \text{ concretions } \pi : \mathcal{SP}_K \to \mathcal{SVC}_K \text{ s.t.}$$
$$\mathcal{S} \models R_K \wedge \mathcal{S} \models \pi(C) : [\![W]\!]^{\pi}_{\mathcal{S}}([\![\pi(pre)]\!]_{\mathcal{S}}) \subseteq [\![\pi(post)]\!]_{\mathcal{S}} \ .$$

There are different ways of proving template correctness. The verification approach introduced in [27] encodes correctness as satisfiability problem. For brevity, we provide a correctness proof for Example 2 in terms of Hoare-style verification. Since our semantics is almost the same as that in [1], we can readily use their proof calculus (augmenting it with rules for **foreach** and service calls; rules omitted here). The proof outline in Fig. 3 shows that, starting from the precondition, the postcondition is reached by the workflow; to do this, we rewrite the **if** without **else** into an **if-then-else** with an empty **else** (skip) construct.

5 Template Instantiation

Templates are used to describe generic forms of service compositions, independent of concrete domains and thus concrete services. To describe templates we employ template ontologies which fix the concepts usable in a template. For instantiation, we replace service placeholders with concrete services, which are typed over concrete domain ontologies. To this end, we define a *mapping* between a template ontology and a domain ontology. While general ontology mapping has to deal with different *ontology conflicts* [22,19], we assume a perfect mapping without conflicts.

Definition 13. *Let $K_T = (C_T, P_T, R_T)$ be a template ontology and $K_D = (C_D, P_D, R_D)$ be a domain ontology. Then $K_T \rhd_f K_D$ is an* 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 $p : T_1 \times \cdots \times T_n \to$ **bool** we have $f_P(p) : f_C(T_1) \times \cdots \times f_C(T_n) \to$ **bool**;*
- *f preserves the rules R_T, that is $\forall r \in R_T$ with $r = b_1 \wedge \cdots \wedge b_n \to h_1 \wedge \cdots \wedge h_m$, there is $r' \in R_D$ with $r' = f(b_1) \wedge \cdots \wedge f(b_n) \to f(h_1) \wedge \cdots \wedge f(h_m)$.*

We assume the mapping pair f to map standard types to themselves, e.g., $f(\textbf{bool}) = \textbf{bool}$, $f(\textbf{set } T) = \textbf{set } (f(T))$. For brevity, we use f as a shorthand notation for the application of the correct mappings f_C, f_P, or rule preservation.

To replace service placeholders (typed over a template ontology K_T) with service descriptions (typed over a domain ontology K_D), we define an concretion π_f for two ontologies with $K_T \rhd_f K_D$.

```
1  {∀a ∈ A : pre^sp_V(a) ∧ A_0 = A}
2  A' := ∅
3  {inv: A' = {a' ∈ A_0 \ A | g(a')}}
4  foreach a ∈ A do
5  |   {A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')}}
6  |   (y) := V(a)
7  |   {post^sp_V(a, y) ∧ A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')}} // service call
8  |   if fp(y) then
9  |   |   {fp(y) ∧ post^sp_V(a, y) ∧ A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')}}
10 |   |   A' := A' ∪ {a}
11 |   |   {g(a) ∧ A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')} ∪ {a}} // constr./set union
12 |   |   {inv}
13 |   else
14 |   |   {¬fp(y) ∧ post^sp_V(a, y) ∧ A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')}}
15 |   |   skip
16 |   |   {¬g(a) ∧ A' = {a' ∈ A_0 \ (A ∪ {a}) | g(a')}} // constraint
17 |   |   {inv}
18 |   fi
19 |   {inv}
20 od
21 {inv ∧ A = ∅} // foreach
22 {A' = {a' ∈ A_0 | g(a')}}
```

Fig. 3. Proof outline for correctness of the filter template; comments refer to the semantics definition

Definition 14. *Let K_T and K_D be ontologies with $K_T \rhd_f K_D$, let $\mathcal{SP}_T$ be the set of service placeholders over K_T, and $\mathcal{SVC}_D$ be the set of service descriptions over K_D. Then $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ is a* concretion of service placeholders from K_T to K_D, *if it respects signatures with respect to f, that is,*

$$\text{if } \pi_f(sp) = svc \text{ and } sp : T_1 \times \cdots \times T_k \to T_{k+1} \times \cdots \times T_N \ ,$$
$$\text{then } svc : f(T_1) \times \cdots \times f(T_k) \to f(T_{k+1}) \times \cdots \times f(T_N) \ .$$

We lift π_f to pre- and postconditions of placeholders, such that $\pi_f(pre^{sp}) \in \Phi_D$ and $free(\pi_f(pre^{sp})) = I_{svc}$, as well as $\pi_f(post^{sp}) \in \Phi_D$ and $free(\pi_f(post^{sp})) = I_{svc} \cup O_{svc}$. In short, π_f maps placeholders to services, using the ontology mapping f of $K_T \rhd_f K_D$ to translate types from the template to the domain ontology.

As the semantics of workflows rely on logical structures, we need to clarify the relation of structures over K_T to structures over K_D: if a structure satisfies the rules of K_D, then there exists a corresponding one satisfying the rules of K_T.

Proposition 1. *Let $K_T = (C_T, P_T, R_T)$ and $K_D = (C_D, P_D, R_D)$ be ontologies and $\mathcal{S} = (\mathcal{U}, \mathcal{I})$ a logical structure over K_D. If $K_T \rhd_f K_D$, and $\mathcal{S} \models R_D$, then we can construct a corresponding logical structure $\mathcal{S}^{\rhd_f}$, where*

$$\mathcal{S}^{\rhd_f} = (\mathcal{U}^{\rhd_f}, \mathcal{I}^{\rhd_f}), \text{ with } \quad \mathcal{U}^{\rhd_f}_T = \mathcal{U}_{f(T)} \text{ and}$$
$$\mathcal{I}^{\rhd_f}(p) = \mathcal{I}(f(p)) \text{ for } p \in \mathcal{P}_T$$

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

Later, we will reason about formulae containing placeholders, which are satisfied by a logical structure, and which follow from the rules of an ontology. Therefore, in addition to the construction of a corresponding logical structure, we construct a corresponding mapping from placeholders to service descriptions as well.

Proposition 2. *Let* $K_T = (C_T, P_T, R_T)$ *and* $K_D = (C_D, P_D, R_D)$ *be ontologies with* $K_T \rhd_f K_D$*, and* $\mathcal{S} = (\mathcal{U}, \mathcal{I})$ *a logical structure over* K_D*. Let* $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ *be a concretion, and* $\Psi \in \Phi_T$ *be a formula containing placeholders from* $\mathcal{SP}_T$*. If* $\mathcal{S}$ *satisfies the rules of* K_D *and the concretized formula* $\pi_f(\Psi)$*, then we can construct a corresponding concretion* $\pi_f^{\rhd_f} : \mathcal{SP}_T \to \mathcal{SVC}_T$ *within the template ontology; let signature names be* $sp \in \mathcal{SP}_T, svc \in \mathcal{SVC}_D, svc' \in \mathcal{SVC}_T$ *and* $\pi_f(sp) = svc$*,* $\pi_f^{\rhd_f}(sp) = svc'$*, and* svc *and* svc' *refer to the same name; if* $\pi_f^{\rhd_f}$ *is signature preserving (Def. 14), and*

$$\pi_f^{\rhd_f}(sp) = svc' \text{ with } svc' : T_1 \times \cdots \times T_j \to T_{j+1} \times \cdots \times T_N \ ,$$

then we also know, that

$$\begin{aligned} \pi_f(sp) = svc \text{ with } svc &: f(T_1) \times \cdots \times f(T_j) \to f(T_{j+1}) \times \cdots \times f(T_N) \\ pre_{svc'} \in \Phi_T &\text{ such that } f(pre_{svc'}) = pre_{svc} \\ post_{svc'} \in \Phi_T &\text{ such that } f(post_{svc'}) = post_{svc} \ ; \end{aligned}$$

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 $\Psi \in \Phi_T$. Then, the interpretations $\mathcal{I}_D$ are fix for every predicate. We can construct $\mathcal{S}^{\rhd_f}$ by Prop. 1, 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 $f(\pi_f^{\rhd_f}(pre_{svc}^{sp})) = \pi_f(pre_{svc}^{sp})$ (same for postcondition). By definition, the interpretations are then mapped to the corresponding predicates, and $\sigma \models_{\mathcal{S}} \pi_f(\Psi) \Rightarrow \sigma \models_{\mathcal{S}^{\rhd_f}} \pi_f^{\rhd_f}(\Psi)$.

The same is true for $\sigma \not\models_{\mathcal{S}} \pi_f(\Psi)$, therefore $\sigma \models_{\mathcal{S}} \pi_f(\Psi) \Leftrightarrow \sigma \models_{\mathcal{S}^{\rhd_f}} \pi_f^{\rhd_f}(\Psi)$. □

We can conclude that the set of states satisfying a formula with instantiated placeholders under a structure $\mathcal{S}$, is the same as for the corresponding $\mathcal{S}^{\rhd_f}$:

Lemma 1. *Let* $K_T = (C_T, P_T, R_T)$ *a template ontology and* $K_D = (C_D, P_D, R_D)$ *a domain ontology with* $K_T \rhd_f K_D$*, a concretion* π_f*, a formula* $F \in \Phi_T$ *containing placeholders from* $\mathcal{SP}_T$*, and a structure* $\mathcal{S} \models \pi_f(F)$*, then*

$$[\![\, \pi_f(F) \,]\!]_{\mathcal{S}} = [\![\, \pi_f^{\rhd_f}(F) \,]\!]_{\mathcal{S}^{\rhd_f}} \ .$$

Note that we do not need to give an interpretation for the standard function and predicate symbols since their interpretation is always the same.

We continue our example with a domain ontology and a service description.

Example 3. Let K_D be an ontology of the (simplified) domain of restaurants with concepts $C_D = \{\mathit{Restaurant}, \mathit{Rating}\}$, predicates $P_D = \{\mathit{isMinRating} : \mathit{Rating} \to \mathbf{bool}, \mathit{goodRestaurant} : \mathit{Restaurant} \to \mathbf{bool}, \mathit{fastFood} : \mathit{Restaurant} \to \mathbf{bool}, \mathit{cheap} : \mathit{Restaurant} \to \mathbf{bool}, \mathit{hasRating} : \mathit{Restaurant} \times \mathit{Rating} \to \mathbf{bool}\}$ and rules[3]

$$\begin{aligned} \mathit{fastFood}(\mathit{res}) &\Rightarrow \mathit{cheap}(\mathit{res}) \\ \mathit{hasRating}(\mathit{res}, \mathit{rat}) \wedge \mathit{isMinRating}(\mathit{rat}) &\Rightarrow \mathit{goodRestaurant}(\mathit{res}) \\ \mathit{hasRating}(\mathit{res}, \mathit{rat}) \wedge \neg \mathit{isMinRating}(\mathit{rat}) &\Rightarrow \neg \mathit{goodRestaurant}(\mathit{res}) \end{aligned}$$

We define a mapping $f = (f_C, f_P)$ from K_T of Example 1 with $K_T \rhd_f K_D$ as:

$$\begin{aligned} &f_C : \mathit{Elem} \mapsto \mathit{Restaurant}, \mathit{Value} \mapsto \mathit{Rating} \\ &f_P : \mathit{fp} \mapsto \mathit{isMinRating}, g \mapsto \mathit{goodRestaurant}\ . \end{aligned}$$

Since the template ontology has no rules, f trivially preserves them. For our restaurant ontology, we assume a service *Vld* to provide a lookup service for ratings of restaurants. It consists of the signature $\mathit{Restaurant} \to \mathit{Rating}$, an input *res*, an output *rat*, precondition $\mathit{pre}_{\mathit{Vld}} = \mathit{true}$ (it provides ratings for all restaurants), and postcondition $\mathit{post}_{\mathit{Vld}} = \mathit{hasRating}(\mathit{res}, \mathit{rat})$ (the returned rating belongs to the input restaurant).

Such services can replace service placeholders in the template. In addition, instantiation requires replacing boolean conditions in the template workflow (because they use template predicates) with their counterparts in the domain ontology. To this end, we apply the ontology mapping f to the boolean conditions.

Definition 15. *Let $WT = (N, I, O, \mathcal{SP}_T, \mathit{pre}, \mathit{post}, C, W)$ be a workflow template over a template ontology K_T, let K_D be a domain ontology with set of services $\mathcal{SVC}_D$ and $K_T \rhd_f K_D$ with $f = (f_C, f_P)$. Let $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ be a concretion of the service placeholders in WT to services of the domain ontology. The* instantiation *of the workflow W with respect to π and f, $\pi_f(W)$, is inductively defined as follows:*

$$\begin{aligned} \pi_f(\text{skip}) &:= \text{ skip} \qquad \pi_f(u := t) := \ u := t \\ \pi_f((u_{j+1}, \ldots, u_k) := \mathit{Svc}(i_1, \ldots, i_j)) &:= \quad (u_{j+1}, \ldots, u_k) := \pi_f(\mathit{Svc})(i_1, \ldots, i_j) \\ \pi_f(W_1; W_2) &:= \quad \pi_f(W_1); \pi_f(W_2) \\ \pi_f(\textbf{if } B \textbf{ then } W_1 \textbf{ else } W_2 \textbf{ fi}) &:= \quad \textbf{if } f(B) \textbf{ then } \pi_f(W_1) \textbf{ else } \pi_f(W_2) \textbf{ fi} \\ \pi_f(\textbf{while } B \textbf{ do } W \textbf{ od}) &:= \quad \textbf{while } f(B) \textbf{ do } \pi_f(W) \textbf{ od} \\ \pi_f(\textbf{foreach } a \in A \textbf{ do } W \textbf{ od}) &:= \quad \textbf{foreach } a \in A \textbf{ do } \pi_f(W) \textbf{ od}\ . \end{aligned}$$

Note that terms t do not need to be mapped by f since they only contain function symbols over standard types.

[3] Ontology languages provide dedicated constructs to specify different properties of predicates, e.g., transitivity or cardinality ("every restaurant has exactly one rating"). These constructs can be expressed using rules, but for simplicity, we omitted them in this example.

Name	: RestaurantFilter
Inputs	: A with $type(A) = \mathbf{set}\ Restaurant$
Outputs	: A' with $type(A') = \mathbf{set}\ Restaurant$
Services	: $Vld : Restaurant \rightarrow Rating$
Precondition	: $\{\forall a \in A : pre_{Vld}(a)\}$
Postcondition	: $\{A' = \{a \in A \mid g(a)\}\}$
Constraints	: $\{\forall x, y : post_{Vld}(x, y) \wedge isMinRating(y) \Rightarrow goodRestaurant(x),$ $\forall x, y : post_{Vld}(x, y) \wedge \neg isMinRating(y) \Rightarrow \neg goodRestaurant(x)\}$

```
1 A' := ∅ ;
2 foreach a ∈ A do
3 |  (y) := Vld(a);
4 |  if isMinRating(y) then  A' := A' ∪ {a} fi ;
5 od
```

Fig. 4. Instantiation of the Filter template with a restaurant ontology and a rating acquisition service

When templates are instantiated, we get *service compositions*. A service composition is a workflow (over a domain ontology) without service placeholders. Figure 4 shows an instantiation of the filter template from Fig. 1, using the ontology mapping of Example 3 and $\pi_f(V) = Vld$.

For the semantics of service compositions, we re-use the semantics definition of templates. This time, however, we can omit the parameter π_f, since all services are concrete. Therefore, the only parameter left for the semantics is the logical structure: the interpretation of domain ontology predicates is still not fixed. The correctness condition can thus directly be re-used, except that service compositions do not come with fixed pre- and postconditions (unlike templates).

Definition 16. *A service composition W over a domain ontology K is* correct with respect to some precondition $pre \in \Phi_K$ and some postcondition $post \in \Phi_K$ *if the following holds:*

$$\forall \text{ logical structures } \mathcal{S} \text{ s.t. } \mathcal{S} \models R_K : [\![\, W \,]\!]_{\mathcal{S}}([\![\, pre \,]\!]_{\mathcal{S}}) \subseteq [\![\, post \,]\!]_{\mathcal{S}} \ .$$

We have defined correctness of templates and service compositions as well as semantics for both. If a template and a composition are typed over the same ontology, we can conclude from the definitions that they have the same semantics.

Lemma 2. *Let WT be a template with workflow W, and $\pi : \mathcal{SP}_K \rightarrow \mathcal{SVC}_K$ be an instantiation with services over ontology K. Then the following holds:*

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

From the semantics and correctness definitions above, and assuming that the template is already proven correct, we state the following: To prove correctness of a service composition, it is sufficient to show that the instantiated constraints of the template can be derived from the rules of the domain ontology.

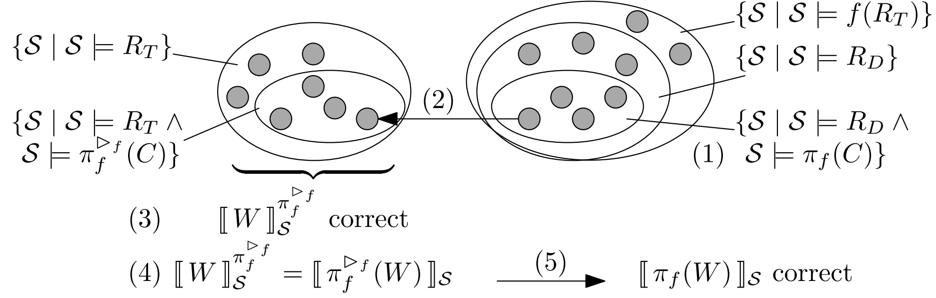

Fig. 5. Overview of the proof of Theorem 1, with main steps (1) to (5)

Theorem 1. *Let $WT = (N, I, O, \mathcal{SP}_T, pre, post, C, W)$ be a correct workflow template over a template ontology K_T, K_D a domain ontology such that $K_T \rhd_f K_D$ with a mapping $f = (f_C, f_P)$, and let $\pi_f : \mathcal{SP}_T \to \mathcal{SVC}_D$ be a concretion of service placeholders in the template with services of the domain ontology.*

If $R_D \models \pi_f(C)$ then $\pi_f(W)$ is correct with respect to $\pi_f(pre)$ and $\pi_f(post)$.

Figure 5 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 will now give the formal proof.

Proof: Let $WT = (N, I, O, \mathcal{SP}_T, pre, post, C, W)$ be a correct workflow template over a template ontology K_T, and $\pi_f(W)$ be a service composition over a domain ontology K_D with $K_T \rhd_f K_D$. We have to show that, for any concretion π_f where the domain ontology satisfies the concretized template constraints, $\pi_f(W)$ is indeed a correct instantiation of template WT, that is, it is correct with respect to the concretized precondition and postcondition of WT. Formally: for all structures S_D and instantiations π_f the following has to hold:

$$\mathcal{S}_D \models R_D \wedge R_D \models \pi_f(C) : [\![\pi_f(W)]\!]_{\mathcal{S}_D}([\![\pi_f(pre)]\!]_{\mathcal{S}_D}) \subseteq [\![\pi_f(post)]\!]_{\mathcal{S}_D} \ .$$

To start, fix $\mathcal{S}_D$ and π_f such that

$$\mathcal{S}_D \models R_D \wedge R_D \models \pi_f(C) \ .$$

If $\mathcal{S}_D$ satisfies its rules, then it satisfies the subset of rules that is the range of the homomorphous mapping:

$$\mathcal{S}_D \models R_D \Rightarrow \mathcal{S}_D \models f(R_T) \text{ because } f(R_T) \subseteq R_D \text{ (by Def. 13).}$$

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

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

of course also holds. Now we "switch" to the template ontology; by $K_T \rhd_f K_D$, we can construct a corresponding structure over the template ontology (Prop. 1) along with a mapping $\pi_f^{\rhd_f} : \mathcal{SP}_T \to \mathcal{SVC}_T$ (Prop. 2):

$$\exists \mathcal{S}^{\rhd_f}, \exists \pi_f^{\rhd_f} : \mathcal{S}^{\rhd_f} \models R_T \wedge \mathcal{S}^{\rhd_f} \models \pi_f^{\rhd_f}(C) \; .$$

By Def. 12, and because WT is correct, we know

$$\exists \mathcal{S}^{\rhd_f}, \exists \pi_f^{\rhd_f} :$$
$$\mathcal{S}^{\rhd_f} \models R_T \wedge \mathcal{S}^{\rhd_f} \models \pi_f^{\rhd_f}(C) : [\![W]\!]^{\pi_f^{\rhd_f}}_{\mathcal{S}^{\rhd_f}} ([\![\pi_f^{\rhd_f}(pre)]\!]_{\mathcal{S}^{\rhd_f}}) \subseteq [\![\pi_f^{\rhd_f}(post)]\!]_{\mathcal{S}^{\rhd_f}} \; .$$

As we are currently solely in the template ontology, by Lemma 2 we know that a template and an instantiation have the same semantics, and therefore

$$\exists \mathcal{S}^{\rhd_f}, \pi_f^{\rhd_f} :$$
$$\mathcal{S}^{\rhd_f} \models R_T \wedge \mathcal{S}^{\rhd_f} \models \pi_f^{\rhd_f}(C) : [\![\pi_f^{\rhd_f}(W)]\!]_{\mathcal{S}^{\rhd_f}} ([\![\pi_f^{\rhd_f}(pre)]\!]_{\mathcal{S}^{\rhd_f}}) \subseteq [\![\pi_f^{\rhd_f}(post)]\!]_{\mathcal{S}^{\rhd_f}} \; .$$

If we "switch back" to the domain ontology, by Lemma 1, we can use our original $\mathcal{S}_D$ again:

$$\mathcal{S}_D \models R_D \wedge \mathcal{S}_D \models \pi_f(C) : [\![\pi_f(W)]\!]_{\mathcal{S}_D} ([\![\pi_f(pre)]\!]_{\mathcal{S}_D}) \subseteq [\![\pi_f(post)]\!]_{\mathcal{S}_D} \; .$$

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

For our example template, we look at the instantiation $\pi_f(V) = \mathit{Vld}$ from Fig. 4. It can easily be shown that the concretized constraints follow from the rules R_D of the restaurant domain. Thus the service composition as given by the instantiated template is correct with respect to the mapped pre- and postconditions of the template which are $\forall a \in A : \mathit{true}$ (precondition) and $A' = \{a \in A \mid \mathit{goodRestaurant}(a)\}$.

6 Discussion

Our approach contains the following aspects: (1) we have correct templates with a formal, parameterized semantics, (2) instantiate them with services of a concrete domain, and (3) show correctness of the instantiation by correctness of simple side-conditions. As instantiated templates come along with a full-fledged service description, they can be treated as services themselves, and therefore be re-used as services in other template instantiations.

On a basic level, verification of service compositions is not fundamentally different from verification of programs. However, especially the context of on-the-fly composition and verification comes with timing constraints; we therefore believe that it is not feasible to prove correctness of service compositions individually. Domain-independent templates can, on the other hand, be verified without

timing constraints (e.g., by dedicated specialists). Then, instantiations can be created on-the-fly, without the necessity of a complete verification: checking the validity of the instantiated constraints is sufficient.

Working with templates in general is common in modular software design methods, component-based software development, and service-oriented architectures, either by dedicated modeling constructs, or by best-practices. However, templates are not necessarily verified.

An early approach of formally specifying a service composition as a parameterized template, and to get provably correct instantiations, is the CARE approach [20,16]. In CARE so-called *fragments* are used for modeling: *primitives*, which come with a black-box description and are proven to be correct externally; and *composites*, which are used to model complex algorithms. The Z modeling language [28] is used as a concise formal notation. For a composition's specification, a *proof obligation* is derived automatically and can be proven by an automated (and/or interactive) theorem prover to show that the instantiation is correct wrt. the requirements. In contrast to CARE, we define correctness for an *incomplete* template, and show that it is sufficient to proof that an instantiation adheres to come constraints, instead of proving correctness for the complete instantiation. Also, we integrate formalized domain knowledge into our approach. [15] also uses the CARE method, but focuses on matching and adaptation.

Based on the development of adaptation techniques, [9] advocates the need for verification at runtime, to verify compositions which changed while already deployed. SimuLizar [6] extends the Palladio Component Model (PCM, [7]) with fuzzy requirements for adaptation using the temporal-logic-based RELAX language, targeting scalability analysis. While both focus on non-functional properties, it would be promising to apply our template- and constraint-based verification to similar runtime contexts.

There is also more research to define formal semantics for existing industrial workflow languages. While [26] defines an event algebra for general workflows, [10] defines a semantic for WS-BPEL [23] based on abstract machines, and [24] based on Petri nets. [18] derives a data flow network from BPEL and translate it to a Promela [17] model.

Also based on generalized data flow networks, the REO approach [3] focuses on communication between entities (e.g., services), by using channel-based communication models, and defining a semantics based on times data streams [2].

While we use a simple imperative programming style language to present our approach, we believe it is possible to apply our results to existing workflow or software architecture languages. To do this, the target language needs a notion of placeholders, and a proper mapping between the languages has to be defined. Our own ongoing experiments are based on an extension of PCM (esp. with pre-/postconditions, [4]), where we also work on SAT-based verification of the instantiation process.

7 Conclusion

In this paper, we presented an approach to create service compositions for different domains by instantiation of domain-independent templates. Moreover, if these templates are provably correct, we have shown that verification of service compositions can be reduced to verification of side-conditions of the instantiation: the instantiated template constraints have to hold in the target domain.

To prove this, we defined an abstract semantics for workflow templates containing service placeholders, which is parameterized with concretions (of placeholders), and logical structures (which fix the concrete meaning). We defined correctness with respect to pre- and postconditions based on this parameterized semantics. If templates formalize instantiation constraints in the form of rules, we have shown that all possible template instantiations are correct, if the (corresponding) instantiation of these constraints are correct.

Therefore, using this approach to create service compositions, their verification can be reduced to verification of side-conditions.

We would like to thank our colleague Felix Mohr for several discussions about the use of templates in service compositions, as well as the anonymous reviewers for their feedback.

References

1. Apt, K., de Boer, F., Olderog, E.R.: Verification of sequential and concurrent programs. Springer (2009)
2. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) Recent Trends in Algebraic Development Techniques, LNCS, vol. 2755, pp. 34–55. Springer (2003)
3. Arbad, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (2004)
4. Arifulina, S., Becker, M., Platenius, M.C., Walther, S.: SeSAME: Modeling and analyzing high-quality service compositions. In: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014), Tool Demonstrations. ACM (15 - 19 Sep 2014), to appear
5. Baader, F., Horrocks, I., Sattler, U.: Description logics. In: Frank van Harmelen, V.L., Porter, B. (eds.) Foundations of Artificial Intelligence, vol. Volume 3, pp. 135–179. Elsevier (2008)
6. Becker, M., Luckey, M., Becker, S.: Performance analysis of self-adaptive systems for requirements validation at design-time. In: QoSA. pp. 43–52. ACM (2013)
7. Becker, S., Koziolek, H., Reussner, R.: 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
8. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: SERA. pp. 40–48 (2006)
9. Calinescu, R., Ghezzi, C., Kwiatkowska, M., Mirandola, R.: Self-adaptive software needs quantitative verification at runtime. Commun. ACM 55(9), 69–77 (2012)
10. Farahbod, R., Glässer, U., Vajihollahi, M.: Abstract operational semantics of the business process execution language for web services. Tech. rep. (2005)

11. Franconi, E., Tessaris, S.: Rules and queries with ontologies: A unified logical framework. In: Ohlbach, H.J., Schaffert, S. (eds.) Principles and Practice of Semantic Web Reasoning, LNCS, vol. 3208, pp. 50–60. Springer (2004)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
13. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5(2), 199–220 (1993)
14. Guarino, N., Oberle, D., Staab, S.: What is an ontology? In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 1–17. International Handbooks on Information Systems, Springer (2009)
15. Hemer, D.: Semi-automated component-based development of formally verified software. Electronic Notes in Theoretical Computer Science 187(0), 173 – 188 (2007), proceedings of the 11th Refinement Workshop (REFINE 2006)
16. Hemer, D., Lindsay, P.: Reuse of verified design templates through extended pattern matching. In: Fitzgerald, J., Jones, C., Lucas, P. (eds.) FME '97: Industrial Applications and Strengthened Foundations of Formal Methods, LNCS, vol. 1313, pp. 495–514. Springer (1997)
17. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
18. Kovács, M., Gönczy, L.: Simulation and formal analysis of workflow models. Electronic Notes in Theoretical Computer Science 211, 221–230 (2008)
19. Kumar, S.K., Harding, J.A.: Ontology mapping using description logic and bridging axioms. Computers in Industry 64(1), 19–28 (2013)
20. Lindsay, P.A., Hemer, D.: An industrial-strength method for the construction of formally verified software. Australian Software Engineering Conference p. 27 (1996)
21. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing semantics to web services: The OWL-S approach. In: Cardoso, J., Sheth, A. (eds.) Semantic Web Services and Web Process Composition, LNCS, vol. 3387, pp. 26–42. Springer (2005)
22. Noy, N.F.: Ontology mapping. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 573–590. International Handbooks on Information Systems, Springer (2009)
23. OASIS: Web services business process execution language v2.0, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
24. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming 67(2–3), 162–198 (2007)
25. Pnueli, A.: The temporal logic of programs. FOCS pp. 46–57 (1977)
26. Singh, M.P.: Formal Aspects of Workflow Management – Part 1: Semantics (1997)
27. Walther, S., Wehrheim, H.: Knowledge-based verification of service compositions – an SMT approach. In: ICECCS. pp. 24–32 (2013)
28. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996)