2013 International Conference on Engineering of Complex Computer Systems

# Knowledge-based Verification of Service Compositions – an SMT Approach

Sven Walther and Heike Wehrheim
Department of Computer Science
University of Paderborn, Germany
{swalther,wehrheim}@mail.upb.de

***Abstract*—In the *Semantic (Web) Services* area, services are considered black boxes with a semantic description of their interfaces as to allow for precise service selection and configuration. The semantic description is usually grounded on domain-specific concepts as modeled in ontologies. This accounts to *types* used in service signatures, but also to *predicates* occurring in preconditions and effects of services. Ontologies, in particular those enhanced with *rules*, capture the knowledge of domain experts on properties of and relations between domain concepts.**

**In this paper, we present a verification technique for service compositions which makes use of this domain knowledge. We consider a service composition to be an assembly of services of which we just know signatures, preconditions, and effects. We aim at proving that a composition satisfies a (user-defined) requirement, specified in terms of guaranteed preconditions and required postconditions. As an underlying verification engine we use an SMT solver. To take advantage of the domain knowledge (and often, to *enable* verification at all), the knowledge is fed into the solver in the form of *sorts*, *uninterpreted functions* and in particular *assertions* as to enhance the solver's reasoning capabilities. Thereby, we allow for deductions within a domain previously unknown to the solver. We exemplify our technique on a case study from the area of water network optimization software.**

## I. Introduction

The concept of service-oriented architectures (SOA) advocates methods and principles for designing applications on the basis of interoperable services, and by this encourages the development of small reusable software services. To allow for precise service selection and configuration, such services are not only described by signature-based APIs but have richer interfaces including protocol and functionality descriptions, i.e., preconditions and effects. This principle in particular manifests itself in the Semantic Web Services initiative [1]. The terms and concepts on which such interface descriptions are based are defined in *ontologies*. Ontologies formalize domain specific knowledge of fundamental concepts and their relations. In particular, ontologies combined with rules (like OWL with SWRL [2]) allow to describe properties of the concepts within the domain which are not expressible with ontologies alone. Thus, a rule enhanced ontology can be used to capture even complex *domain knowledge*.

While there is a large amount of research on using ontologies for service selection and matching (like [3], [4], [5]), or for reasoning about ontologies themselves (e.g., consistency checking, querying, or integration [6]), this is not the case for service composition *verification*. In this paper, we deal with the latter. We assume to be given a composition of services,

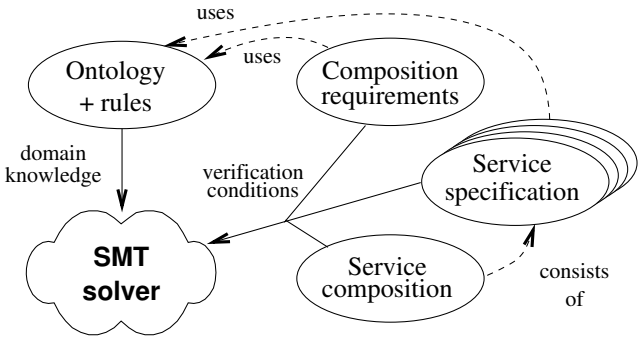



Fig. 1. Overview of our approach

each with an ontology-based interface specification. Apart from interfaces, nothing is known about the services (black-box view). In a SOA context, this is a quite likely scenario: providers sell their services but not the code itself. In fact, a service might not even run on the consumer side, but could either completely stay on the provider machine or run in the cloud. Furthermore, the requirements on an assembled service composition are specific to the domain; instead of proving general safety or reachability properties alone (as state-of-the-art software verification tools do), consumers expect the verification to prove domain-specific requirements. We leverage this by grounding service specifications on ontologies.

As an example, we introduce our case study "water supply network optimization". Software services in this domain handle different tasks of analyzing and optimizing existing municipal water supply networks. Single services designed for different sub-tasks can be assembled into a composition. This concerns services like (a) compacting the size (and layout) of network models, (b) generating mathematical optimization problems from networks, (c) solving optimization problems, and (d) applying optimal solutions to networks. As the behavior of these services is specified in terms of interfaces only, this is a black-box view and for the analysis we can therefore only assume that services adhere to their specification.

Every water network has specific hydraulic characteristics, as well as other properties like cost of operation. A typical domain specific requirement on a composition of some optimization services is that an optimized network (produced by the composition) has the *same* hydraulic characteristics as the original input network, but *better* (e.g., lower) cost of operation.

Our approach to the verification of such service compo-

978-0-7695-5007-7/13 $26.00 © 2013 IEEE
DOI 10.1109/ICECCS.2013.14

sition is based on the use of an SMT solver (satisfiability modulo theories solver [7]) as reasoning engine. Basically, our technique feeds three types of inputs (domain knowledge, service interface specifications, and assembly) into the SMT solver in different forms (see Fig. 1). These inputs, combined with the user's requirements specification, are encoded as first-order logic formulae. As we want to show that the requirements always hold, some of the formulae are negated, and the negation is required to be unsatisfiable (that is, no counterexample can be found). In that case, the composition meets the requirements. More specifically, we start with an ontology of the domain which models – beside the standard concepts and their relations – additional rules about the domain by first-order logic. The predicates therein are the relations in the ontology. Providers of services use the ontology to specify a service's signature and its preconditions and effects. Consumers use the ontology to specify requirements of a service composition. For verification, we use the concepts of the ontology as *types* for the solver (in our case Z3 [8]), relations as *uninterpreted functions*, and rules as constraints on the interpretation of these functions. The rules are thus being used for deduction together with the decidable theories of the solver (e.g., linear arithmetic). The creation of verification conditions for a given service composition and requirements follows ideas of Hoare-style proofs [9]. In our example we verify an application consisting of an assembly of services in a loop by using loop invariants and termination functions. It turns out that the verification requires the additional domain knowledge for a successful reasoning: the knowledge of human domain experts (e.g. about hydraulic properties of different forms of networks) needs to be provided to the solver.

The remaining paper is structured as follows. The next section introduces our running example, as well as an overview on ontologies, service specifications, and requirements. Section III introduces rules as an extension to ontologies, and section IV explains our approach of translating the specification into a SAT/SMT problem. The last section presents related work and concludes.

## II. Background

We start by introducing our example domain and giving an overview of the basic concepts we use in our approach. Our example comes from the domain of operative planning of water supply networks, but our approach is more general: the scenario we envision[1] is that of a consumer stating requirements about a desired application, a configuration process proposing different possible compositions of small services based on the services' signature only, and the verification process validating these compositions against the requirements of the consumer. Here, the mutual basis is that services are small, specified as black boxes, and combined to a larger composition that has to meet some requirements. The validation part of such a composition is the topic of this paper.

After introducing our example domain, this section gives a basic introduction to ontologies as the core of the semantic web. Then we present service specifications, service compositions, and requirements.

[1]See the Collaborative Research Centre 901 – On-The-Fly Computing, http://sfb901.uni-paderborn.de.

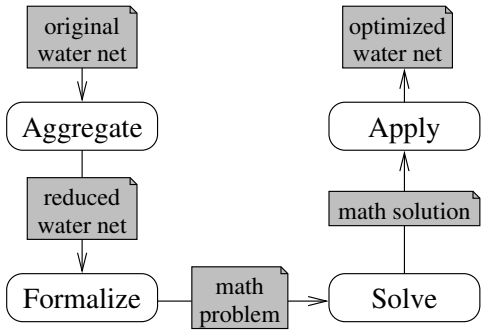



Fig. 2. Abstract workflow of optimizing water supply networks

### A. Example Domain

In the domain of operative planning of (drinking) water networks, an important task is to analyze and optimize water networks towards a specific goal, e.g., operation cost minimization, pump switching, or replacement of network elements (like tanks). Sound mathematical models for water networks become very complex, and are based on nonlinear programming (NLP), or even nonlinear mixed-integer programming (MINLP). Therefore, a common step in computing a network optimization is to simplify the network model. The complexity of a model is *reduced* by *aggregating* certain network structures into simpler ones (examples below). The resulting model has to be hydraulically equivalent to the input model, but due to its reduced complexity the goal of computing an optimization problem becomes feasible for NLP or MINLP solvers[2] [10].

The reduction of network models is the first step to solve the optimization problem. The overall workflow of a network's optimization consists of four main steps: (a) aggregate a network model to reduce its complexity and size, (b) formalize the network model and the specific optimization goal as a mathematical problem, (c) use an off-the-shelf solver to solve this problem, and (d) apply the mathematical solution to the network model. Fig. 2 depicts this workflow. Every service as specified using preconditions and effects. The aggregation part itself can be seen as another service composition. To demonstrate our own verification approach, we focus on this part, as it will contain a loop, which requires additional proof strategies (Section IV).

### B. Ontologies

Ontologies provide means to formalize knowledge of a specific domain. Here, the domain is *water supply networks*. An ontology acts as a *parameter* for the various steps of our approach, namely for providers specifying their services, consumers stating requirements, and finally the verification.

Domain knowledge is separated into the main parts of *concepts* and *roles* (or relations) between them. Besides a graphical representation, as used in this paper, a description-logics- or first-order-logic-style representation is also common ([11]). Concepts (or classes) of an ontology denote sets of individuals that share some common features, while roles are used to describe *properties* of individuals of a concept.

[2]For a list of off-the-shelf solvers, see http://www.gams.com/solvers.

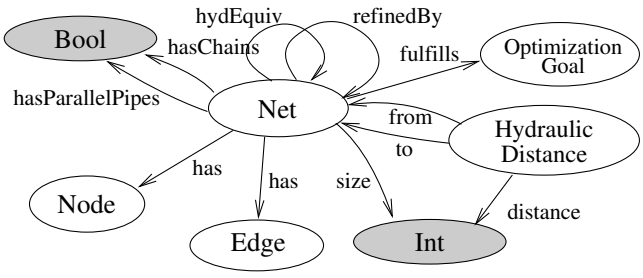



Fig. 3. Simplified version of a water supply network domain ontology

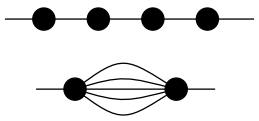


Fig. 4. A node chain (above) and five parallel pipes (below)

Concepts are typically mapped to *types* that are used in operation signatures of service descriptions.

Fig. 3 shows an extract of our example domain ontology. It includes the concepts *Net* (which describes networks in general), and *OptimizationGoal* (which describes in which way a network shall be changed). *RefinedBy* is a role, relating a net to another one if there is some sort of structural refinement between them. The *hasChains* and *hasParallelPipes* roles (also related to a net's structure) describe properties that map individuals of the concept *Net* to individuals of the basic type *Bool*.

While relations in ontologies are always binary, it is possible to model $n$-ary relations as binary ones by treating a specific $n$-ary relation as a new concept: for our example we assume that two water networks have a specific *hydraulic distance*, which we consider to be an integer value. As we want to relate three parameters (two nets and a number), we consider hydraulic distance as a separate concept which has roles *from* and *to* to the compared nets, and *distance* as a data property of the basic type *Int* (integer).

We refrain from giving a complete ontology for water supply network optimization here. The depicted concepts and roles are the ones needed to demonstrate our approach. Ontologies are typically formalized using the Web Ontology Language (OWL, [12]).

### C. Services and Service Compositions

The ontology is the basis for specifying single services (or better, their interfaces) out of which complex aggregations can be formed. In the domain of network optimization there are for instance several aggregation services available, among others two services called *RemoveChains* and *RemoveParallelPipes*. Both change the structure (or layout) of a net. Sequences of nodes are called *chains*, and multiple pipes (edges) linking the same nodes directly are called *parallel pipes* (Fig. 4). The two services eliminate chains and parallel pipes, respectively, from a net. Fig. 5 shows a simple example net, where chains (node sequences) and parallel pipes are repeatedly replaced by simple pipes (edges).

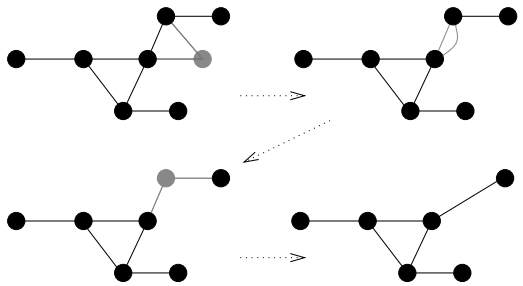


Fig. 5. Repeatedly removing *chains* and *parallel pipes* from a water network graph. Chains (nodes with two edges) and parallel pipes (parallel edges) are marked gray

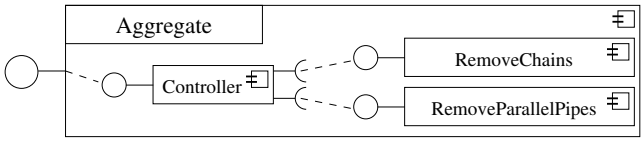



Fig. 6. Service composition containing two aggregation services (*RemoveChains, RemoveParallelPipes*)

We assume services like *RemoveChains* to be specified by interfaces that contain operation signatures. An operation signature contains a name, and names and types of input and output parameters. Following the semantic (web) service approach, we also require preconditions and effects for each signature.

For sake of readability, we assume that every service has exactly one operation (named uniquely), and we give specifications using an abstract notation instead of OWL. Preconditions and effects are given in first-order logic notation. The following table lists the specification of two aggregation services, as potentially supplied by a provider.

| **Service: RemoveChains** | |
|---|---|
| *input:* | *i: Net* |
| *output:* | *o: Net* |
| *pre:* | $hasChains(i)$ |
| *effect:* | $hydEquiv(i,o) \wedge size(o) < size(i)$ |
| | $\wedge\ refinedBy(i,o) \wedge \neg hasChains(o)$ |
| **Service: RemoveParallelPipes** | |
| *input:* | *i: Net* |
| *output:* | *o: Net* |
| *pre:* | $hasParallelPipes(i)$ |
| *effect:* | $hydEquiv(i,o) \wedge size(o) < size(i)$ |
| | $\wedge\ refinedBy(i,o) \wedge \neg hasParallelPipes(o)$ |

A configuration process might combine the services *RemoveChains* and *RemoveParallelPipes* into a service composition *Aggregate*. In our setting, service compositions will be given by a structure diagram (Fig. 6) combined with an activity diagram (Fig. 7) to specify the control flow. Every composition has a coordinating service (named *Controller*) which calls the other services as specified in the activity diagram. We assume that the composition *Aggregate* from Fig. 6 is itself used

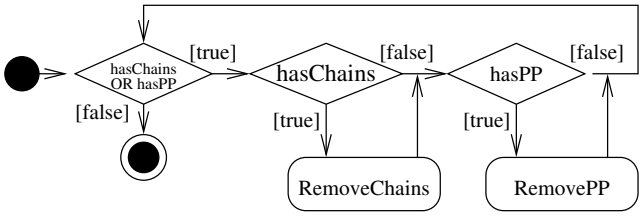



Fig. 7. Activity diagram of the *Aggregate* composition (joining diamonds omitted for brevity)

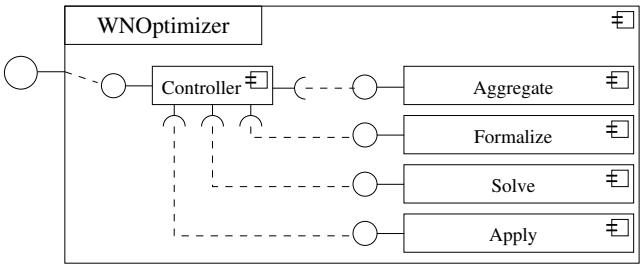



Fig. 8. Service composition containing several services to handle the optimization workflow

by a composition *WNOptimizer*, which handles the complete optimization process (Fig. 8).

In general, activity diagrams contain a start and stop state. We use a simple box with a service name to indicate the call to a service. *Conditions* (diamonds) are used to describe alternative control flows or loops. In our activity diagram the services *RemoveChains* and *RemoveParallelPipes* are continously called as long as the loop condition $c = hasChains \vee hasParallelPipes$ holds. As the effect of *RemoveParallelPipes* does not guarantee anything concerning chains, it is possible that new chains are introduced. This is indeed the case: Fig. 5 shows a network that initially contains chains, but no parallel pipes. Removing the chains (by replacing the node with a new pipe) introduces parallel pipes. Replacing parallel pipes with a single pipe, may introduce a new chain. Therefore repeated calls to the services are needed for achieving a chain- and parallel-pipe-free network at the end.

### D. Requirements

To request a composition, the consumer formulates requirements by (again) giving a signature (consisting of input and output parameter names and types) as well as *preconditions* and *effects*, which represent the requested composition. Preconditions describe the valid input to the composition, and effects describe the required output. We use first-order logic formulae (based on concepts and/or roles of the domain ontology) to formalize preconditions and effects.

In the domain of water network optimization, a consumer would typically require to get compositions which are able to compute an optimized water net given some input net. Thus the requested composition will get an input net and has to return an output net. The requirements will however differ in their optimization goals: some consumers might want to have a net with operation costs below a threshold, or a net not having tanks that are not fully utilized. Most of the time there is in addition a requirement stating that input and output net must have the same or similar hydraulic characteristics. Different requirements necessitate different compositions. The purpose of our technique is to check whether a given composition adheres to some given requirements.

In our example, the aggregation service can be seen as a composition as well, consisting of two services *RemoveChains* and *RemoveParallelPipes* (Fig. 6). Here, the requirement might be as follows: the input consists of a net containing chains, or parallel pipes, or both (precondition), and the output is required to not contain any chains or parallel pipes anymore (effect). To simulate an allowed error margin, the output net is furthermore required to have a hydraulic distance to the input of less than 5. As an abbreviation, we use the notation

$$HydDistance(n_1, n_2) < x$$

to specify a hydraulic distance of less than $x$ between nets $n_1$ and $n_2$. These requirements can be formalized in the following way:

| **Requirements** (for aggregation services) | |
|---|---|
| *input:* | *i: Net* |
| *output:* | *o: Net* |
| *pre:* | $hasChains(i) \vee hasParallelPipes(i)$ |
| *effect:* | $HydDistance(i, o) < 5 \wedge refinedBy(i, o)$ $\wedge \neg hasChains(o) \wedge \neg hasParallelPipes(o)$ |

Now, we have specified (domain specific) requirements for an aggregation service composition, and we are given a candidate composition. The task is now to show that the candidate composition fulfills these requirements using all of our available domain knowledge. The reader may have already noticed that different predicates are used to formulate the effects of the services and the requirements (and this will frequently be the case as consumer and producer will not be the same person): while the services guarantee *hydraulic equivalence* (using the predicate $hydEquiv$), the requirements ask for a maximum hydraulic distance of 5. While domain experts will easily see the connections between these two predicates, our ontology does not contain this information yet (and consequently our reasoning technique cannot use it). In fact, normal ontologies do not provide flexible means to relate such predicates. The next section introduces *rules* to enable formalizing additional knowledge which is not expressible with ontologies alone.

## III. Ontologies with Rules

The last remark already shows that we are not yet at the end of specifying domain knowledge. The two aggregation services guarantee the output net to be hydraulically equivalent to the input net, and the user requires a maximum hydraulic distance of 5. The fact that hydraulic equivalence implies a hydraulic distance of zero (which is clear to domain experts) is not yet formalized. To utilize this knowledge for reasoning, we need ways of specifying all sorts of constraints for our domain concepts and roles, in particular by *relating* different *roles*.

Using only ontology languages, it is not always possible to formalize all types of constraints. Languages like the *Resource Description Framework* (RDF, [13]) only support concepts

and (binary) roles between them. RDF Schema [14] adds predefined roles which enable a reasoner to derive additional knowledge (in terms of assertions, that have been known only implicitly), but there is no way to create complex relations between roles. The direct semantics of OWL 2 is (roughly) compatible to the description logic $\mathcal{SROIQ}$ [15], [11], and therefore provide additional concept constructors similar to set-theoretic operations. It also provides predefined statements to add information about roles, like constraining cardinalities, inversion, transitivity, symmetry, or functionality. Our running example includes the following properties (stating only those used in reasoning):

| **Properties** (employed in reasoning) | |
|---|---|
| $isRefinedBy$ | reflexive, transitive, anti-symmetric |
| $hydEquiv$ | reflexive, transitive, symmetric |

However, while OWL enables us to formalize such predefined characteristics, this is not sufficient. It is, for instance, possible to create concept expressions based on intersection or union, but nothing similar is possible for roles. Therefore additional means to formalize *rules* are needed [16]. Rules in general relate atoms and predicates. They consist of an *antecedent* $A$ and a *consequent* $C$ and are written as an implication

$$A \rightarrow C.$$

A rule that contains a conjunction as a consequent can be split into different rules with one predicate as a consequent for each. A rule which has no antecedent is a *fact*, as the consequent has to hold all the time. Applied to our running example, rules allow us to add further information. We want to state that hydraulic equivalence results in a hydraulic difference of 0, and vice versa (here $i, o$ are nets and implicitly universally quantified). We furthermore need to state that nets have a minimum size of 0:

| **Rules** (employed in reasoning) | |
|---|---|
| $hydEquiv(i,o) \Rightarrow$ | $HydDistance(i,o) = 0$ |
| $HydDistance(i,o) = 0 \Rightarrow$ | $hydEquiv(i,o)$ |
| | $size(n) \geq 0$ |

Now, we have introduced all necessary domain knowledge, formalized by an ontology and rules, and its use to specify requirements and services. While there are several languages to specify services, compositions, and requirements (like PCM [17], UML [18], WSDL [19]), our verification approach outlined in the next section does not depend on the exact languages used for specification. Whether or not a specific language is suitable, depends on its expressiveness.

## IV. Verification via SMT Solving

The ontology with its properties and rules formalize the domain knowledge consisting of concepts, roles between them, and additional constraints. The types used in services and requirements specifications stem from the same ontology.[3]

[3]This is in fact not strictly necessary; one could use ontology matching and consistency techniques to allow for different ontologies [20].

```
1  (declare-sort Net)
2  (declare-fun hasChains (Net) Bool)
3  (declare-fun hasParallelPipes (Net) Bool)
4  (declare-fun size (Net) Int)
5
6  ; All networks have at least size zero
7  (assert
8    (forall ((n Net))
9      (≥ (size n) 0))
10 )
11
12 ; Hydraulic equivalence and distance of zero are the same
13 (assert
14   (forall ((g Net) (h Net))
15   (⇒ (hydEquiv g h)
16       (= (HydDistance g h) 0)
17   ))
18 )
19 (assert
20   (forall ((g Net) (h Net))
21   (⇒ (= (HydDistance g h) 0)
22       (hydEquiv g h)
23   ))
24 )
```

Fig. 9. Translation of (part of) the rules

Together with the service composition model this is the input for the verification procedure. The verification consists of three steps: first, the concepts of the ontology have to be translated into input for the SMT solver. This is only done once for every ontology. Second, we need to create a first order formula describing the service composition's behavior. And finally, we need to derive verification conditions for the correctness check of the service compositions with respect to the consumer's requirements. The last two steps need to be carried out for every pair of requirements and service composition which is to be verified.

We start with the translation of the domain knowledge captured by the ontology with rules. The SMT solver we currently use is Z3 [8]. The basic scheme here is as follows: we use concepts of the ontology as *sorts* (types) for the solver, roles as *uninterpreted functions*[4], and properties (like transitivity) and rules will be *assertions* about the uninterpreted functions that give constraints on their interpretation. As an example consider the Z3 code in Fig. 9. In line 1 we declare a sort `Net` for networks, and in lines 2, 3 and 4 the relations `hasChains`, `hasParallelPipes` and `size`. Lines 6 to 24 capture some of the rules about these concepts and predicates: every network has at least size zero, and hydraulic equivalence (*hydEquiv*) is the same as a hydraulic distance of zero (*HydDistance*). In a similar way we translate the other concepts and rules to capture the domain knowledge.

The second step is the translation of the service composition itself. We are given a structure diagram consisting of a number of services and in particular a main service. The main service contains a behavior description (activity diagram) describing how the other services are being called. In our example composition the top service has a single input

[4]As services are considered as black boxes, this is sufficient. Proving that a single service adheres to its own specification is not the topic of this paper.

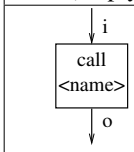
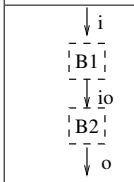
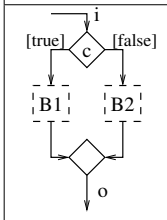
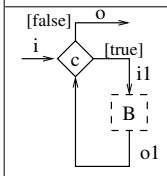
| Block type | Formula |
|---|---|
| ◉ (empty) | $i = o$ |
|  | $pre_{name}(i) \Rightarrow \mathit{effect}_{name}(i, o)$ |
|  | $\exists io : \varphi_{B_1}(i, io) \wedge \varphi_{B_2}(io, o)$ |
|  | $(c(i) \wedge \varphi_{B_1}(i, o))$<br>$\vee\ (\neg c(i) \wedge \varphi_{B_2}(i, o))$ |
|  | $\neg c(o) \wedge ((\neg c(i) \wedge i = o)$<br>$\vee\ c(i) \wedge p(i, o))$ |

Fig. 10. Translation of activity diagram elements

which is forwarded to the first called sub-service and the services' outputs are also simply given as input to the next one. However, it is straightforward to extend our translation concept to include more than one parameter. The translation is defined by induction on the structure of the activity diagram and is sketched in Fig. 10.

In the activity diagram the following structures are allowed and have the following translations into first-order logic. We always generate a formula $\varphi(i, o)$ which has the variables $i$ (input) and $o$ (output) as the only free variables. Thereby we get a formula which describes the behavior in terms of an input-output relationship.

Empty block
: The empty block (induction start) is translated into the formula $i = o$; nothing happens and thus input equals output.

Service call
: Given a service *name* with precondition $pre_{name}(x)$ and effect $\mathit{effect}_{name}(x, y)$, then the formula for the call is

$$\varphi_{call} := pre_{name}(i) \Rightarrow \mathit{effect}_{name}(i, o)$$

(if the preconditions holds, the effect can be guaranteed).

Sequence
: Assuming that the formulae of two blocks $B_1$ and $B_2$ are $\varphi_{B_1}$ and $\varphi_{B_2}$, respectively, the formula for the sequence is

$$\varphi_{seq} := \exists io : \varphi_{B_1}(i, io) \wedge \varphi_{B_2}(io, o).$$

Conditional
: Again assuming that the formulae of two blocks $B_1$ and $B_2$ are $\varphi_{B_1}$ and $\varphi_{B_2}$, respectively, and the condition is $c(i)$, the formula for the conditional (if-then-else) is

$$\varphi_{ite} := (c(i) \wedge \varphi_{B_1}(i, o)) \vee (\neg c(i) \wedge \varphi_{B_2}(i, o)).$$

Fig. 7 shows two examples of conditional calls within a sequence.

Loop
: Assume that we are given a loop construct with condition $c(i)$ and loop body $B$ with associated formula $\varphi_B$ (which is needed only to create verification conditions, see later). The translation here is a bit more difficult since we do not know the exact number of loop executions. A possible approach is to make a bounded unfolding of the loop body for user given bounds. Here, we use *loop invariants* instead (see below for the verification condition associated to invariants). The loop invariant is used to capture the behavior of the entire loop (to the extent necessary for verification). Given a loop invariant $p(i, o)$ on the inputs and outputs, we generate the formula

$$\varphi_{loop} := \neg c(o) \wedge \big((\neg c(i) \wedge o = i) \vee (c(i) \wedge p(i, o))\big)$$

(after the loop, the loop condition is false, and either the loop condition is not true on the input and then input equals output, or the loop is executed (at least once) and thereby the invariant established). Note that the loop invariant relates inputs and outputs to/from the loop (in contrast to normal state-based loop invariants). Fig. 7 shows the example loop containing an complex inner block.

This translation gives us a formula $\varphi_{act}$ for an activity diagram of a service composition. For our example we have the following parts. The loop invariant is

$$p(i, o) := \mathit{refinedBy}(i, o) \wedge \mathit{hydEquiv}(i, o).$$

The invariant states that after every loop execution the output network remains hydraulic equivalent to the input and is a refinement of it. The resulting formula of the composition used for *Aggregate* thus is:

$$\begin{aligned}\varphi_{act}(i, o) := \neg\big(&\mathit{hasChains}(o) \vee \mathit{hasParallelPipes}(o)\big) \\ \wedge\ \Big(&\big(\neg\big(\mathit{hasChains}(i) \vee \mathit{hasParallelPipes}(i)\big) \\ &\wedge (i = o)\big) \\ \vee\ &\big(\big(\mathit{hasChains}(i) \vee \mathit{hasParallelPipes}(i)\big) \\ &\wedge \mathit{refinedBy}(i, o) \wedge \mathit{hydEquiv}(i, o)\big)\Big)\end{aligned}$$

Note that we do not see the services in the loop body in here since these are covered by the loop invariant. Of course we need to show that $p$ is indeed an invariant (see next).

In addition to the service composition we are given the consumer's requirements with precondition $pre_{Req}$ and effect $effect_{Req}$. The last step is the generation of a number of verification conditions. The verification conditions need to capture three aspects: (1) the service composition guarantees the requirements, (2) the loop invariant is actually an invariant, and (3) the loop terminates (if we are interested in total and not just partial correctness). For the first aspect we need to check validity of the following implication:

$$\forall i, o : pre_{Req}(i) \wedge \varphi_{act}(i, o) \Rightarrow effect_{Req}(i, o) \quad \text{(VC-REQ)}$$

In Z3 we do so by checking satisfiability of the negation of the above formula. If this formula is unsatisfiable, validity holds, as no counterexample was found. Taking the above formula $\varphi_{act}$ and our requirements this is indeed true.

The second and third type of verification conditions concern loops. We have used a loop invariant to capture the behavior of the loop, and thus we need to show that the loop invariant is actually an invariant of the loop. The non-standard part in our approach is the fact that loop invariants need to talk about the input-output behavior of a loop instead of just properties on states. To do so, we use loop invariants of type $p(i, o)$. We thus need to show that (a) the invariant is established by one loop execution, and (b) once established is preserved by further executions. Formally, we need to show the following (where $LoopBody$ is the loop body):

$$\forall i, o : c(i) \wedge \varphi_{LoopBody}(i, o) \Rightarrow p(i, o) \quad \text{(VC-INV1)}$$

$$\forall i, k, o : p(i, k) \wedge c(k) \wedge p(k, o) \Rightarrow p(i, o) \quad \text{(VC-INV2)}$$

VC-INV1 shows the establishment of an invariant $p$. VC-INV2 shows transitivity of $p$. From both, we can deduce that $p$ is preserved by further loop executions.

The final part is termination. To this we need to find a termination function $t$ on the variables of the loop, and show that (a) $t$ is decreased with every loop execution, and (b) $t$ is always greater than or equal to zero. In terms of formulae we check validity of:

$$\forall i, o, z : c(i) \wedge t(i) = z \wedge \varphi_{LoopBody}(i, o) \Rightarrow t(o) < z \quad \text{(VC-TERM1)}$$

$$\forall i : t(i) \geq 0 \quad \text{(VC-TERM2)}$$

In our example, the termination function $t$ corresponds to *size*. To give an impression of how to state these verification conditions within Z3, Fig. 11 shows the Z3 code of the verification condition for termination.

So far, both loop invariants and termination functions have to be found by hand. We are however confident that template-based techniques for deriving invariants (like [21]) can be fruitfully applied in our setting. For instance, a natural candidate for a loop invariant is a predicate appearing in the effects of all services in the loop body. Once we have derived a number of invariant candidates, we can then automatically check their usefulness and validity with Z3.

```
1  (define-fun LoopTermination1 () Bool
2    (forall ((i Net) (o Net) (z Int))
3      (⇒ (and (C i)
4              (= (T i) z)
5              (LoopBody i o))
6         (< (T o) z))))
7
8  (assert (not LoopTermination1))
9  (check-sat)
```

Fig. 11. Verifying VC-TERM1 for the loop

For our example, all of the verification conditions are satisfied. However, if we remove some of our domain knowledge, we will fail to prove the verification conditions. As an example: for verifying termination (in particular VC-TERM2) it is crucial to know that all networks have a size greater or equal to zero. For verification of the requirements (VC-REQ), the rules that relate hydraulic equivalence and hydraulic similarity are necessary. If these rules are removed, Z3 will find an interpretation of the uninterpreted functions which makes the negated formulae satisfiable. These interpretations (or models) can serve as counterexamples for the domain experts: when they are no valid interpretations from the point of view of the domain specialist, then there are additional rules and properties needed to eliminate them.

All in all, we have thus obtained a verification procedure for service compositions in arbitrary domains, in which the domain knowledge as given by an ontology is used for reasoning within the solver. In this paper, we demonstrated our approach on a service composition for water network aggregation, as the control flow of this composition contains a loop. However, we applied our technique to the complete optimization composition (which includes the aggregation composition) as well.

## V. Conclusion

The semantic web makes extensive use of ontologies to provide a rich description of services. Part of this description is the specification of preconditions and effects of a service call. While different approaches like SWRL [2] exist to make such specifications possible, proposals like OWL-S [22], that integrate ontologies and Web-service specifications, are not standardized yet. There exist approaches to use rules for ontology reasoning [23], [24] and service matching [25], and [26] uses ontology and rules reasoning in the context of (Web-) service verification, but only for reasoning about correctness of their knowledge base. The use of ontological knowledge for verification of service compositions, based on preconditions and effects, remains open.

In this paper, we have presented an approach to make use of a rich semantic service description in verification. To this end, we process the knowledge base of a specific domain and derive types and predicates from an ontology with rules. Using these predicates, we create first-order logic formulae and proof

| | t | result |
|---|---|---|
| Aggregation only (as in this paper) | 37 | unsat |
| ... without knowledge of minimal size of nets | 41 | sat |
| ... without knowledge of hyd. property relations | 43 | sat |
| Complete optimization composition | 38 | unsat |
| ... without knowledge of minimal size of nets | 45 | sat |
| ... without knowledge of hyd. property relations | 53 | sat |

Fig. 12. Results of Z3 solver runs (average of 1000 runs for each scenario) on a common laptop (Intel i3-370M 2.4 GHz, 4 GB RAM). Time (t) in milliseconds (rounded), based on Z3's own statistics.

conditions based on the preconditions and effects of services and the control flow information of a service composition. Formalized as a satisfiability problem, we use the formulae as input to an SMT solver, to check whether or not the composition meets the requirements.

Up to now, we derived formulae and SMT code manually. We used Z3 [8] as an SMT solver. Currently we are working on an automated SMT code generation, that works on ontologies specified with OWL and SWRL, and an extended version of the Palladio Component Model [17] as a specification language for services and service compositions. Solving first-order logic satisfiability problems is undecidable in principle. However, we think that our approach is feasible for application to the verification of Web-service compositions. The scenario presented in this paper (aggregation only, and aggregation embedded into the workflow as depicted in Fig. 2) was verified fast (Fig. 12). In this scenario, the number of services used in a composition is yet relatively small. Future work includes the evaluation of our verification procedure for larger examples from different domains, as well as integration with other semantic web-based techniques like matching.

**Related work.** While ontology-based reasoning is a well-understood topic, and several specialized reasoners exist (e.g., FaCT++ [27]), in the first place they reason about the validity and soundness of a given ontology. There are approaches to address the problem of ontology reasoning with automated theorem proving [28], [29], as well as with SAT/SMT solving [30]. Both concentrate on the soundness of ontologies, and do not verify service compositions. There are, however, early approaches of analyzing service compositions based on DAML+OIL (a predecessor of OWL) using petri nets [31]. [32] uses Spin to verify Web-service interactions specified in OWL-S. [33] also uses OWL-S to specify Web-services, but they use user-preference based HTN planning to create compositions. [34] advocates service composition and verification relying on linear programming, based on a simplified service specification without ontologies. [35] combines Web-service compositions (given as BPEL models) and service descriptions (using visual contracts), but is specifically concerned with checking that preconditions of services hold at their point of execution in a business process. [36] verifies Web-service compositions (specified in a CSP-style language) based on trace inclusion (using the PAT framework [37]), but there is no relation to ontological knowledge.

## ACKNOWLEDGMENTS

We like to thank our domain experts Florian Stapel and Corinna Hallmann for numerous discussions on water supply networks.

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

## REFERENCES

[1] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. McGuinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with OWL-S," *World Wide Web*, vol. 10, pp. 243–277, 2007.

[2] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: a Semantic Web Rule Language combining OWL and RuleML," 2004, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[3] L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology," in *WWW '03*. ACM, 2003, pp. 331–339.

[4] Z. Huma, C. Gerth, G. Engels, and O. Juwig, "Towards an Automatic Service Discovery for UML-Based Rich Service Descriptions," in *MoDELS*, 2012, pp. 709–725.

[5] M. C. Jaeger, G. Rojec-Goldmann, C. Liebetruth, G. Mühl, and K. Geihs, "Ranked Matching for Service Descriptions Using OWL-S," in *KiVS*, 2005, pp. 91–102.

[6] K. Baclawski, M. Kokar, R. Waldinger, and P. Kogut, "Consistency checking of semantic web ontologies," in *The Semantic Web - ISWC 2002*, ser. LNCS. Springer, 2002, vol. 2342, pp. 454–459.

[7] D. Kroening and O. Strichman, "A framework for satisfiability modulo theories," *Formal Aspects of Computing*, vol. 21, pp. 485–494, 2009.

[8] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008, pp. 337–340.

[9] K. Apt, F. de Boer, and E.-R. Olderog, *Verification of sequential and concurrent programs*. Springer, 2009.

[10] J. Burgschweiger, B. Gnädig, and M. Steinbach, "Nonlinear programming techniques for operative planning in large drinking water networks," *Open Applied Mathematics Journal*, vol. 3, pp. 14–28, 2009.

[11] F. Baader, I. Horrocks, and U. Sattler, "Chapter 3 description logics," in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, V. L. Frank van Harmelen and B. Porter, Eds. Elsevier, 2008, vol. 3, pp. 135 – 179.

[12] M. Schneider, "OWL 2 Web Ontology Language RDF-based semantics," W3C, W3C Recommendation, Oct. 2009, http://www.w3.org/TR/2009/REC-owl2-rdf-based-semantics-20091027/.

[13] D. Beckett, "RDF/XML syntax specification (revised)," W3C, W3C Recommendation, Feb. 2004, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[14] R. V. Guha and D. Brickley, "RDF vocabulary description language 1.0: RDF schema," W3C, W3C Recommendation, Feb. 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[15] P. F. Patel-Schneider, B. Motik, and B. C. Grau, "OWL 2 web ontology language direct semantics," W3C, W3C Recommendation, Oct. 2009, http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/.

[16] E. Franconi and S. Tessaris, "Rules and queries with ontologies: A unified logical framework," in *Principles and Practice of Semantic Web Reasoning*, ser. LNCS, H. Ohlbach and S. Schaffert, Eds. Springer, 2004, vol. 3208, pp. 50–60.

[17] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009.

[18] "Unified modeling language (UML)," OMG, 2011, version 2.4.1.

[19] R. Chinnici, S. Weerawarana, J.-J. Moreau, and A. Ryman, "Web services description language (WSDL) version 2.0 part 1: Core language," W3C, W3C Recommendation, Jun. 2007, http://www.w3.org/TR/2007/REC-wsdl20-20070626.

[20] N. F. Noy, "Ontology mapping," in *Handbook on Ontologies*, S. Staab and R. Studer, Eds. Springer, 2009, pp. 573–590.

[21] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *SIGPLAN*, ser. PLDI '09. New York: ACM, 2009, pp. 223–234.

[22] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic markup for web services," 2004, http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.

[23] W. Drabent, T. Eiter, G. Ianni, T. Krennwallner, T. Lukasiewicz, and J. Maluszynski, "Hybrid reasoning with rules and ontologies," in *REWERSE*, 2009, pp. 1–49.

[24] J. Baumeister, T. Kleemann, and D. Seipel, "Towards the verification of ontologies with rules," in *FLAIRS Conference*, 2007, pp. 524–529.

[25] A. B. Bener, V. Ozadali, and E. S. Ilhan, "Semantic matchmaker with precondition and effect matching using SWRL," *Expert Syst. Appl.*, vol. 36, no. 5, pp. 9371–9377, Jul. 2009.

[26] Z. Xu and J. Wu, "Ontology reasoning and services composition verification towards o-rgps requirement meta-model," in *ICACTE*, 2010, pp. 273–277.

[27] "FaCT++ OWL2 reasoner," http://owl.cs.manchester.ac.uk/fact++/.

[28] I. Horrocks and A. Voronkov, "Reasoning support for expressive ontology languages using a theorem prover," in *FoIKS*. Springer, 2006, pp. 201–218.

[29] M. Schneider and G. Sutcliffe, "Reasoning in the OWL 2 full ontology language using first-order automated theorem proving," in *Automated Deduction – CADE-23*, ser. LNCS, N. Bjørner and V. Sofronie-Stokkermans, Eds. Springer, 2011, vol. 6803, pp. 461–475.

[30] M. Vescovi, "Exploiting SAT and SMT techniques for automated reasoning and ontology manipulation in description logics," Ph.D. dissertation, University of Trento, 2011.

[31] S. Narayanan and S. McIlraith, "Simulation, verification and automated composition of web services," in *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*. ACM, 2002, pp. 77–88.

[32] A. Ankolekar, M. Paolucci, and K. Sycara, "Towards a formal verification of OWL-S process models," in *The Semantic Web – ISWC 2005*, ser. LNCS, Y. Gil, E. Motta, V. Benjamins, and M. Musen, Eds. Springer, 2005, vol. 3729, pp. 37–51.

[33] S. Sohrabi and S. McIlraith, "Preference-based web service composition: A middle ground between execution and search," in *The Semantic Web – ISWC 2010*, ser. LNCS, P. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Pan, I. Horrocks, and B. Glimm, Eds. Springer, 2010, vol. 6496, pp. 713–729.

[34] L. Cavallaro, E. D. Nitto, C. A. Furia, and M. Pradella, "A tile-based approach for self-assembling service compositions," in *ICECCS*, 2010, pp. 43–52.

[35] G. Engels, B. Güldali, C. Soltenborn, and H. Wehrheim, "Assuring consistency of business process models and web services using visual contracts," in *AGTIVE*, 2007, pp. 17–31.

[36] J. Sun, Y. Liu, J. S. Dong, G. Pu, and T. H. Tan, "Model-based methods for linking web service choreography and orchestration," in *APSEC 2010*, 2010, pp. 166 – 175.

[37] Y. Liu, J. Sun, and J. S. Dong, "Pat 3: An extensible architecture for building multi-domain model checkers," *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pp. 190–199, 2011.