Towards SMT-based Abstract Planning in PlanICS Ontology*

Artur Niewiadomski[1] and Wojciech Penczek[1,2]
[1]*ICS, Siedlce University, 3-Maja 54, 08-110 Siedlce, Poland*
[2]*ICS, Polish Academy of Sciences, Jana Kazimierza 5, 01-248 Warsaw, Poland*

Keywords: Web Service Composition, SMT, Abstract Planning, Service-oriented Architecture, PlanICS.

Abstract: The paper deals with the abstract planning problem – the first stage of Web Service Composition (WSC) in the PlanICS framework. We present a solution based on a compact representation of abstract plans by multisets of service types and a reduction of the planning problem to a task for an SMT-solver. The paper presents theoretical aspects of the abstract planning as well as some details of our symbolic encoding, followed by preliminary experimental results.

1 INTRODUCTION

The main concept of Service-Oriented Architecture (SOA) (Bell, 2008) consists in using independent components available via well-defined interfaces. Often, a simple web service does not realize the user objective, so a composition of them need to be executed to this aim. The problem of finding such a composition is hard and known as the Web Service Composition Problem (WSCP) (Bell, 2008; Ambroszkiewicz, 2004; Rao and Su, 2004). In this paper, we follow the approach of our system PlanICS (Doliwa et al., 2011), which has been inspired by (Ambroszkiewicz, 2004).

The main assumption is that all the web services in the domain of interest as well as the objects which are processed by the services, can be strictly classified in a hierarchy of *classes*, organised in an *ontology*. Another key idea is to divide planning into several stages. The first phase deals with *classes of services*, where each class represents a set of real-world services, while the second one works in the space of *concrete services*. The first stage produces an *abstract plan*, which becomes a *concrete plan* in the second phase. Such an approach enables to reduce the number of concrete services to be considered. This paper focuses on the abstract planning problem only.

We propose a novel approach based on an application of SMT-solvers. Contrary to a number of other approaches (see Section 1.1), we focus not only on searching for a single solution, but we attempt to find all *significantly different* plans. We start with defining the abstract planning problem (APP) and showing that it is NP-hard. Then, we present a fully original solution of APP based on a compact representation of abstract plans by multisets of service types and a reduction to a task for an SMT-solver, which is the main contribution of our paper. The encoding of blocking formulas allows for pruning the search space with many sequences which use the same service types as some plan already generated. Note that a multiset of size k can be linearised even in $k!$ ways if all its elements are different. To the best of our knowledge, the above approach is novel, and as our experiments show it is also very promising.

The rest of the paper is organized as follows. Related work is discussed in Sect. 1.1. Sect. 2 deals with the abstract planning problem. Sect, 3 presents the SMT-based encoding and implementation of our approach. Sect. 4 discusses experimental results of our planning system. The last section summarizes and discusses the results.

1.1 Related Work

Web services are widely used to implement SOA paradigm, but much of their benefits is revealed when they can be composed automatically. The existing solutions to WSCP are divided into several groups. Following (Li et al., 2010) our approach belongs to AI planning methods, including also approaches based on: automata theory (Mitra et al., 2007), Petri nets (Gehlot and Edupuganti, 2009), theorem proving (Rao et al., 2006), and model checking (Traverso and Pistore, 2004).

*This work has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.

Niewiadomski A. and Penczek W..
Towards SMT-based Abstract Planning in PlanICS Ontology.
DOI: 10.5220/0004514901230131
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2013), pages 123-131
ISBN: 978-989-8565-81-5

A composition method closest to ours is presented in (Nam et al., 2008), where the authors reduce WSCP to a reachability problem of a state-transition system. The problem is encoded as a propositional formula and tested for satisfiability using a SAT-solver. This approach makes use of an ontology describing a hierarchy of types and deals with an inheritance relation. However, we consider also the states of the objects, while (Nam et al., 2008) deals with their types only. Moreover, among other differences, we use a multiset-based SMT encoding instead of SAT.

Most of the applications of SMT in the domain of WSC is related to the automatic verification and testing. For example, a message race detection problem is investigated in (Elwakil et al., 2010), the authors of (Bentakouk et al., 2011) take advantage of symbolic testing and execution techniques in order to check behavioural conformance of WS-BPEL specifications, (Bersani et al., 2010) deals with a service substitutability problem, while (Monakova et al., 2009) exploits SMT to verification of WS-BPEL specifications against business rules. However, to our best knowledge, there are no other approaches dealing with SMT as an engine to WSC.

2 ABSTRACT PLANNING PHASE

APP makes intensive use of the *service types* and the *object types* defined in the *ontology*. A service type represents a set of web services with similar capabilities, while the object types are used to represent data processed by the services. The *attributes* are components of the object types, and the *objects* are simply instances of the object types. An *object state* is determined by its attribute values. However, for APP it is enough to know only whether an attribute does have some value or it does not, and therefore, we introduce the concept of *abstract values*.

Attributes, Object Types, and Objects Let $\mathbb{I}$ denote the set of all *identifiers* used as the names of the types, the objects, and the attributes. In APP we deal with abstract values only, the types of the attributes are irrelevant, and we identify the attributes with their names. Moreover, we denote the set of all attributes by $\mathbb{A}$, where $\mathbb{A} \subset \mathbb{I}$. An *object type* is a pair $(t, Attr)$, where $t \in \mathbb{I}$, and $Attr \subseteq \mathbb{A}$. That is, an object type consists of a type name and a set of attributes. The set of all object types is denoted by $\mathbb{P}$. We define also a transitive, irreflexive, and antisymmetric *inheritance* relation $Ext \subseteq \mathbb{P} \times \mathbb{P}$, such that $\big((t_1, A_1), (t_2, A_2)\big) \in Ext$ iff $t_1 \neq t_2$ and $A_1 \subseteq A_2$. That is, a subtype contains all the attributes of a base type and optionally introduces more attributes. An *object* o is a pair $o = (id, type)$, where $id \in \mathbb{I}$ and $type \in \mathbb{P}$. That is, an object is a pair of the object name, and the object type, denoted by $type(o)$ for a given object o. The set of all objects is denoted by $\mathbb{O}$. Moreover, we define the function $attr : \mathbb{O} \longmapsto 2^{\mathbb{A}}$ returning a set of the attributes for each object of $\mathbb{O}$.

Service Types and User Queries. The service types available for composition are defined in the ontology by *service type specifications*. The user goal is provided in a form of a *user query specification*. Before APP, all the specifications are reduced to sets of objects and *abstract formulas* over them, to be defined in what follows.

Definition 1 (Abstract Formulas)**.** *An abstract formula over a set of objects O and their attributes is defined by the following BNF grammar:*

```
<form> ::= <disj>
<disj> ::= <conj>|<conj> or <disj>
<conj> ::= <lit>|<conj> and <lit>
<lit>  ::= isSet(o.a)|isNull(o.a)|true|false
```

where $O \subseteq \mathbb{O}$, $o \in O$, $a \in attr(o)$, and $o.a$ denotes the attribute a of the object o.

The above grammar defines DNF formulas without negations, i.e., alternatives of clauses, referred to as *abstract clauses*. Every abstract clause is the conjunction of literals, specifying abstract values of object attributes using the functions *isSet* and *isNull*. In the abstract formulas used in APP, we assume that no abstract clause contains both $isSet(o.a)$ and $isNull(o.a)$, for the same $o \in O$ and $a \in attr(o)$. The syntax of the specifications of the user queries and of the service types is the same and it is defined below.

Definition 2 (Specification)**.** *A specification is a 5-tuple $(in, inout, out, pre, post)$, where in, inout, out are pairwise disjoint sets of objects, and pre is an abstract formula defined over objects from $in \cup inout$, while post is an abstract formula defined over objects from $in \cup inout \cup out$.*

In what follows a user query specification q or a service type specification s is denoted by $spec_x = (in_x, inout_x, out_x, pre_x, post_x)$, where $x \in \{q, s\}$, resp. Notice that the objects of in_x are read-only, these of $inout_x$ can change their states, while out_x is to contain new objects only. In order to formally define the *user queries* and the *service types*, which are interpretations of their specifications, we first need to introduce the notions of *valuation functions* and *worlds*.

Definition 3 (Valuations of Object Attributes)**.** *Let φ be an abstract formula over $\mathbb{O}$ s.t. $\varphi = \bigvee_{i=1..n} \alpha_i$, where $n \in \mathbb{N}$, and each α_i is an abstract clause. A valuation of the object attributes over α_i is the partial*

function $v_{\alpha_i} : \bigcup_{o\in\mathbb{O}}\{o\} \times attr(o) \longmapsto \{\textbf{\textit{true}},\textbf{\textit{false}}\}$, *where:*

- $v_{\alpha_i}(o,a)$=***true*** *if* $isSet(o.a)$ *is a literal of* α_i, *or*
- $v_{\alpha_i}(o,a)$=***false*** *if* $isNull(o.a)$ *is a lit. of* α_i, *or*
- $v_{\alpha_i}(o,a)$ *is undefined, otherwise.*

We define the restriction of a valuation function v_{α_i} to a set of objects $O \subset \mathbb{O}$ as $v_{\alpha_i}(O) = v_{\alpha_i}\big|_{\bigcup_{o\in O}\{o\}\times attr(o)}$. We write $v_{\alpha_i}(o)$ instead of $v_{\alpha_i}(\{o\})$, when we restrict the valuation function v_{α_i} to a single object and its attributes.

The undefined values appear when the interpreted abstract formula does not specify abstract values of some attributes. Obviously, this is a typical case in the WSC domain as we often deal with incomplete, uncertain, or irrelevant information. The undefined values are a way to overcome this problem, but they are also a form of representing families of total valuation functions, which is explained below.

Definition 4 (Consistent Functions)**.** *Let A, A′, B be sets such that* $A' \subseteq A$, $f : A \longmapsto B$ *be a total function, and* $f' : A \longmapsto B$ *be a partial function, such that* f' *restricted to* A' *is total. We say that* f is consistent *with* f', *if* f' *restricted to* A' *equals to* f, *i.e.,* $\forall_{a\in A'}\ f'(a) = f(a)$.

Next, for a partial valuation function f, by $total(f)$ we denote the family of the total valuation functions, which are consistent with f. Moreover, we define a *family of the valuation functions* $\mathcal{V}_\varphi$ over the abstract formula φ as the union of the sets of the consistent valuation functions over every abstract clause α_i, i.e., $\mathcal{V}_\varphi = \bigcup_{i=1}^{n} total(v_{\alpha_i})$. The restriction of the family of functions $\mathcal{V}_\varphi$ to a set of objects O and their attributes is defined as $\mathcal{V}_\varphi(O) = \bigcup_{i=1}^{n} total(v_{\alpha_i}(O))$.

Definition 5 (Worlds)**.** *A world w is a pair* $(O_w, v(O_w))$, *where* $O_w \subseteq \mathbb{O}$ *and* $v(O_w)$ *is a total valuation function, restricted to the objects from* O_w, *for some valuation function v. The* size *of w, denoted by* $|w|$, *is the number of the objects in w, i.e.,* $|w| = |O_w|$.

That is, a world represents a state of a set of objects, where each attribute is either set or null.

By a *sub-world* of w we mean a world built from a subset of O_w and v_w restricted to the objects from the chosen subset. Moreover, a pair consisting of a set of objects and a family of total valuation functions defines a *set of worlds*. That is, if $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a family of total valuation functions and $O \subseteq \mathbb{O}$ is a set of objects, then $\big(O, \mathcal{V}(O)\big)$ means the set $\{\big(O, v_i(O)\big) \mid 1 \le i \le n\}$, for $n \in \mathbb{N}$. Finally, the set of all worlds is denoted by $\mathbb{W}$.

Now, we are in a position to define a *service type* and a *user query* as an interpretation of its specification.

Definition 6 (Interpretation of a Specification)**.** *Let* $spec_x = (in_x, inout_x, out_x, pre_x, post_x)$ *be a user query or a service type specification, where* $x \in \{q, s\}$, *resp. An interpretation of* $spec_x$ *is a pair of world sets* $x = (W^x_{pre}, W^x_{post})$, *where:*

- $W^x_{pre} = \big(in_x \cup inout_x, \mathcal{V}^x_{pre}\big)$, *where* $\mathcal{V}^x_{pre}$ *is the family of the valuation functions over* pre_x,
- $W^x_{post} = \big(in_x \cup inout_x \cup out_x, \mathcal{V}^x_{post}\big)$, *where* $\mathcal{V}^x_{post}$ *is the family of the valuation functions over* $post_x$.

An interpretation of a user query (service type) specification is called simply a user query *(*service type*, resp.).*

For a service type (W^s_{pre}, W^s_{post}), W^s_{pre} is called the *input world set*, while W^s_{post} - the *output world set*. The set of all the service types defined in the ontology is denoted by $\mathbb{S}$. For a user query (W^q_{pre}, W^q_{post}), W^q_{pre} is called the *initial world set*, while W^q_{post} - the *expected world set*, and denoted by W^q_{init} and W^q_{exp}, respectively. Notice that out_x is supposed to contain only new objects, which are absent in W^x_{pre}, but present in W^q_{post}. In case of a service type s, the objects of out_s are produced as the result of a *world transformation* to be defined in Sec. 2.2.

2.1 Abstract Planning Overview

Overall, the main goal of APP is to find a composition of service types satisfying a user query, which specifies some initial and some expected worlds. Intuitively, an initial world contains the objects owned by the user, whereas an expected world consists of the objects required to be the result of the service composition. In order to formally define how this is achieved, we need to introduce several auxiliary concepts.

Definition 7 (Compatible Object States)**.** *Let* $o, o' \in \mathbb{O}$, *and let v and* v' *be valuation functions. We say that* $v'(o')$ *is compatible with* $v(o)$, *denoted by* $v'(o') \succ^{obj} v(o)$, *iff:*

- *the types of both objects are the same, or the type of* o' *is a subtype of type of o, i.e.,* $type(o) = type(o')$ *or* $(type(o'), type(o)) \in Ext$, *and*
- *for all attributes of o, we have that* v' *agrees with v, i.e.,* $\forall_{a\in attr(o)} v'(o', a) = v(o, a)$.

Intuitively, an object of a richer type (o') is compatible with the one of a base type (o), provided that the valuations of all common attributes are equal.

Definition 8 (Worlds Compatibility)**.** *Let* $w, w' \in \mathbb{W}$ *be worlds, and let* $w = (O, v)$, *and* $w' = (O', v')$. *We say that the world* w' *is compatible with the world w, denoted by* $w' \succ^{wrl} w$, *iff there exists a*

one-to-one mapping $map : O \longmapsto O'$ *such that* $\forall_{o \in O} v'(map(o)) \succ^{obj} v(o)$.

Intuitively, w' is compatible with w if both of them contain the same number of objects and for each object from w there exists a compatible object in w'.

Definition 9 (Worlds Sub-compatibility)**.** *Let* w, w' *be worlds such that* $w = (O, v)$ *and* $w' = (O', v')$*. The world* w' *is called* sub-compatible *with the world* w*, denoted by* $w' \succ^{swrl} w$ *iff there exists a sub-world of* w' *compatible with* w*.*

2.2 World Transformations

One of the fundamental concepts in our approach concerns a world transformation. A world w, called a *world before*, can be transformed by a service type s, having specification $spec_s$, if w is sub-compatible with some input world of s. The result of such a transformation is a world w', called a *world after*, in which the objects of out_s appear, and, as well as the objects of $inout_s$, they are in the states consistent with some output world of s. The other objects of w do not change their states. In a general case, there may exist a number of worlds possible to obtain after a transformation of a given world by a given service type, because more than one sub-world of w can be compatible with an input world of s. Therefore, we introduce a *context function*, which provides a strict mapping between objects from the worlds before and after, and the objects from the input and output worlds of a service type s.

Definition 10 (Context Function)**.** *A context function* $ctx^s_O : in_s \cup inout_s \cup out_s \longmapsto O$ *is an injection, which for a given service type* s *and a set of objects* O *assigns an object from* O *to each object from* in_s*,* $inout_s$*, and* out_s*.*

Now, we can define a world transformation.

Definition 11 (World Transformation)**.** *Let* $w, w' \in \mathbb{W}$ *be worlds, called a* world before *and a* world after*, respectively, and* $s = (W^s_{pre}, W^s_{post})$ *be a service type. Assume that* $w = (O, v)$*,* $w' = (O', v')$*, where* $O \subseteq O' \subseteq \mathbb{O}$*, and* v, v' *are valuation functions. Let* $ctx^s_{O'}$ *be a context function, and the sets IN, IO, OU be the* $ctx^s_{O'}$ *images of the sets* in_s*,* $inout_s$*, and* out_s*, respect., i.e.,* $IN = ctx^s_{O'}(in_s)$*,* $IO = ctx^s_{O'}(inout_s)$*, and* $OU = ctx^s_{O'}(out_s)$*. Moreover, let* $IN, IO \subseteq (O \cap O')$ *and* $OU = (O' \setminus O)$*.*

We say that a service type s transforms *the world* w *into* w' in the context $ctx^s_{O'}$*, denoted by* $w \xrightarrow{s, ctx^s_{O'}} w'$*, if for some* $v^s_{pre} \in \mathcal{V}^s_{pre}$ *and* $v^s_{post} \in \mathcal{V}^s_{post}$*, all the following conditions hold:*

1. $\big(IN, v(IN)\big) \succ^{wrl} \big(in_s, v^s_{pre}(in_s)\big)$,
2. $(IO, v(IO)) \succ^{wrl} \big(inout_s, v^s_{pre}(inout_s)\big)$,
3. $(IO, v'(IO)) \succ^{wrl} \big(inout_s, v^s_{post}(inout_s)\big)$,
4. $(OU, v'(OU)) \succ^{wrl} \big(out_s, v^s_{post}(out_s)\big)$,
5. $\forall_{o \in (O \setminus IO)}\ \forall_{a \in attr(o)}\ v(o, a) = v'(o, a)$.

Intuitively, (1) the *world before* contains a sub-world built over IN, which is compatible with a sub-world of some input world of the service type s, built over the objects from in_s. (2) The *world before* contains a sub-world built over IO, which is compatible with a sub-world of the input world of the service type s, built over the objects from $inout_s$. (3) After the transformation the state of objects from IO is consistent with $post_s$. (4) The objects produced during the transformation (OU) are in a state consistent with $post_s$. (5) The objects from IN and the objects not involved in the transformation do not change their states.

Definition 12 (Transformation Sequences)**.** *Let* $seq = \big((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k})\big)$ *be a sequence of length* k*, where, for* $1 \leq i \leq k$*,* $s_i \in \mathbb{S}$*,* $O_i \subseteq \mathbb{O}$*, and* $ctx^{s_i}_{O_i}$ *is a context function. We say that a world* w_0 is transformed by the sequence *seq into a world* w_k*, denoted by* $w_0 \overset{seq}{\rightsquigarrow} w_k$*, iff there exists a sequence of worlds* $(w_1, w_2, \ldots, w_{k-1})$ *such that* $\forall_{1 \leq i \leq k}\ w_{i-1} \xrightarrow{s_i, ctx^{s_i}_{O_i}} w_i = (O_i, v_i)$ *for some* v_i*.*

A sequence seq is called a transformation sequence*, if there are two worlds* $w, w' \in \mathbb{W}$ *such that* w *is transformed by seq into* w'*, i.e.,* $w \overset{seq}{\rightsquigarrow} w'$*. The set of all the transformation sequences is denoted by* $\vec{\mathbb{S}}$*.*

Having the transformation sequences defined, we introduce the concept of *user query solutions* or simply *solutions*, in order to define a plan.

Definition 13 (User Query Solution)**.** *Let seq be a transformation sequence and* $q = (W^q_{init}, W^q_{exp})$ *be a user query. We say that seq is a solution of q, if for* $w \in W^q_{init}$ *and some world* w' *such that* $w \overset{seq}{\rightsquigarrow} w'$*, we have* $w' \succ^{swrl} w^q_{exp}$*, for some* $w^q_{exp} \in W^q_{exp}$*. The set of all the solutions of the user query q is denoted by* $QS(q)$*.*

Intuitively, by a solution of q we mean every transformation sequence transforming some initial world of q, to a world sub-compatible to some expected world of q.

2.3 Plans

Basing on the definition of a solution to the user query q, we can now define the concept of an (abstract) plan, by which we mean a non-empty set of solutions of q. We define a plan as an equivalence class of the solutions, which do not differ in the service types used.

The idea is that we do not want to distinguish between solutions composed of the same service types, which differ only in the ordering of their occurrences. So we group them into the same class. There are clearly two motivations behind that. Firstly, the user is typically not interested in obtaining many very similar solutions. Secondly, from the efficiency point of view, the number of equivalence classes can be exponentially smaller than the number of the solutions. To this aim, we introduce an equivalence relation partitioning the set of all the solutions into distinct plans.

Definition 14 (Equivalence of User Query Solutions)**.** *Let the function count* $: \vec{\mathbb{S}} \times \mathbb{S} \longmapsto \mathbb{N}$ *be such that* $count(seq,s)$ *returns the number of occurrences of the service type* s *in the transformation sequence* seq*. The equivalence relation* $\sim \subseteq QS(q) \times QS(q)$ *is defined as follows:* $seq \sim seq'$ *iff* $count(seq,s) = count(seq',s)$ *for each* $s \in \mathbb{S}$*.*

Intuitively, two user query solutions are equivalent if they consist of the same number of the same service types, regardless of the contexts.

Definition 15 (Abstract Plans)**.** *Let* $seq \in QS(q)$ *be a solution of some user query* q*. An abstract plan is a set of all the solutions equivalent to* seq*, i.e., it is equal to* $[seq]_{\sim}$*.*

It is important to notice that all the solutions within an abstract plan are built over the same *multiset* of service types. The following result shows that APP is a hard problem.

Theorem 1. *The abstract planning problem is NP-hard.*

Proof Sketch. Let P be a set of propositions. We define a translation of 3-SAT to the problem of existence of a user query solution. Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$, where each clause $\varphi_i = l^i_1 \vee l^i_2 \vee l^i_3$ with $l^i_j = p$ or $l^i_j = \neg p$ for $p \in P$. We build an ontology of service types ST and a user query q such that there is a plan for q over ST iff φ is satisfiable. $ST = \bigcup_{i=1}^{n} \{s^i_1, s^i_2, s^i_3\}, spec_{s^i_j} = \{in = \emptyset, inout = \{(o,O)\}, out = \{(o_i,O_i)\}, pre = post = isSet(o.p)$ if $l^i_j = p; pre = post = isNull(o.p)$ if $l^i_j = \neg p\}, spec_q = \{in = \emptyset, inout = \{(o,O)\}, out = \{(o_i,O_i) \mid 1 \leq i \leq n\}, pre = post = true\}$. The object o of type O has the attribute $o.p$ for each propositional variable p used in φ. It is easy to show that there is a user solution of q over ST iff there is a valuation satisfying φ. □

Below, we present an example showing an ontology, a user query, and several solutions.

Example 1. *Assume that Selling (S), Transport (T), and Assembly (A) are service types, while Boards, Nails, and Doghouse are object types extending the object type Ware. Because of space limit, we do not give the attributes and the pre and post conditions. S is able to provide any Ware (*$out_S = \{$*(w, Ware)*$\}$*), T can deliver any Ware to the requested destination (*$inout_T = \{$*(w, Ware)*$\}$*), and A can build a doghouse using nails and boards (*$inout_A = \{$*(b, Boards), (n, Nails)*$\}$*,* $out_A = \{$*(d, Doghouse)*$\}$*). The goal is to get a doghouse, (the user query q:* $in_q = inout_q = \emptyset$*,* $out_q = \{(d_1$*, Doghouse)*$\}$*). The shortest solution to q is* (S,T)*. This is the only solution of the plan represented by the multiset* $[S,T]$*. Another solution is* (S,T,S,T,A)*, where the first pair* (S,T) *provides and transports boards while the second pair* (S,T) *provides and delivers nails, which are finally assembled by A. This solution gives another abstract plan represented by* $[A,S,S,T,T]$*. Note that there exists another equivalent solution, namely,* (S,S,T,T,A)*.*

3 SMT-BASED SYMBOLIC ENCODING

This section presents a symbolic encoding of APP by an SMT formula, which is then tested for satisfiability by an SMT-solver. First, we give an overview of our planning algorithm, and discuss the structure of the formula φ^q_k encoding APP. Then, we present the symbolic representation of the objects and the worlds, followed by a sketch of the encoding of selected components of φ^q_k.

3.1 Abstract Planning Algorithm

Given an ontology, a user query, and parameters k_{min} and k_{max}, the planner is searching for solutions of length k such that $k_{min} \leq k \leq k_{max}$. The algorithm begins with $k = k_{min}$ and is looking for a user query solution of length k, by checking the satisfiability of a formula encoding APP.

When the solver returns *SAT*, this means that a solution has been found. This solution is then analysed as a representative of some abstract plan. As a result, a *blocking formula* is computed, which is used to exclude from a further search all the solutions belonging to this abstract plan. If the solver returns *UNSAT*, then there is no more plans of length k. If k does not exceed k_{max}, then k is increased by 1, the new step of the composition is encoded, and the search continues for a possibly longer plan, until k_{max} is reached. Overall, to find a plan of length k satisfying the query q, we build the following SMT-formula φ^q_k:

$$\varphi^q_k = I^q \bigwedge_{i=1..k} \bigvee_{s \in \mathbb{S}} \mathcal{T}^s_i \wedge \mathcal{E}^q_k \wedge \mathcal{B}^q_k, \qquad (1)$$

where I^q and $\mathcal{E}^q_k$ are formulas encoding the initial and the expected worlds, resp., $\mathcal{T}^s_i$ encodes a transformation of one world into another by a service type s, and $\mathcal{B}^q_k$ is a blocking formula.

3.2 Objects and Worlds

The objects and the worlds are represented by sets of *variables*, which are first allocated in the memory of an SMT-solver, and then used to build formulas mentioned in the previous subsection. The representation of an object is called a *symbolic object*. It consists of an integer variable representing the type of an object, called a *type variable*, and a number of Boolean variables to represent the object attributes, called the *attribute variables*. In order to represent all types and identifiers as numbers, we introduce a function $num : \mathbb{A} \cup \mathbb{P} \cup \mathbb{S} \cup \mathbb{O} \longmapsto \mathbb{N}$, which with every attribute, object type, service type, and object assigns a natural number.

A *symbolic world* consists of a number of symbolic objects. Each symbolic world is indexed by a natural number from 0 to k. Formally, the i-th symbolic object from the j-th symbolic world is a tuple: $\mathbf{o}_{i,j} = (\mathbf{t}_{i,j}, \mathbf{a}_{i,0,j}, \mathbf{a}_{i,1,j}, ..., \mathbf{a}_{i,max_{at}-1,j})$, where $\mathbf{t}_{i,j}$ is the type variable, $\mathbf{a}_{i,x,j}$ is the attribute variable for $0 \leq x < max_{at}$, where max_{at} is the maximal number of the attribute variables needed to represent the object. Note that actually a symbolic world represents *a set of worlds*, and only a *valuation* of its variables makes a single world. The j-th symbolic world is denoted by $\mathbf{w}_j$, while the number of the symbolic objects in $\mathbf{w}_j$ - by $|\mathbf{w}_j|$. Fig. 1 shows subsequent symbolic worlds of a transformation sequence.

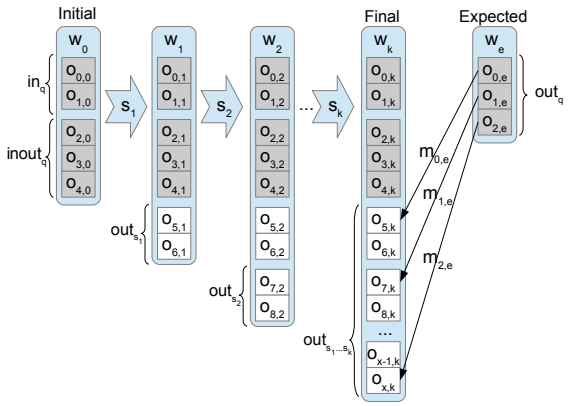

Figure 1: Symbolic worlds.

3.3 User Query

In order to encode the set W^q_{init} by a symbolic world $\mathbf{w}_0$, we allocate the variables needed to represent the objects from $in_q \cup inout_q$. Then, we build the formula I^q over these variables, which encodes the types and the states of the objects from the initial worlds:

$$I^q = tpF\left(\mathbf{w}_0, in_q \cup inout_q\right) \wedge stF\left(\mathbf{w}_0, W^q_{init}\right) \quad (2)$$

The formula $tpF(\mathbf{w}_i, O)$ encoding the types of the objects O over a symbolic world $\mathbf{w}_i$, is defined as:

$$tpF(\mathbf{w}_i, O) = \bigwedge_{o \in O} \mathbf{t}_{num(o),i} = num\big(type(o)\big)$$

The formula $stF(\mathbf{w}_i, W)$ encodes the states of the objects from the worlds $W = (O, \mathcal{V})$ over the symbolic world $\mathbf{w}_i$:

$$stF(\mathbf{w}_i, W) = \bigvee_{v \in \mathcal{V}} \bigwedge_{o \in O} \bigwedge_{a \in attr(o)} vF(\mathbf{w}_i, v, o, a),$$

where $vF(\mathbf{w}_i, v, o, a)$ is the expression encoding the valuation v of the attribute $o.a$ over the variables of the symbolic world $\mathbf{w}_i$, defined as follows:

$$vF(\mathbf{w}_i, v, o, a) = \begin{cases} \mathbf{a}_{num(o),num(a),i}, & \text{if } v(o,a) = \textbf{true}, \\ \neg\mathbf{a}_{num(o),num(a),i}, & \text{if } v(o,a) = \textbf{false}, \\ \textbf{true}, & \text{if } v(o,a) \text{ is undef.} \end{cases}$$

Thus, the symbolic world $\mathbf{w}_0$ represents the initial worlds. Then, after the first transformation we obtain the symbolic world $\mathbf{w}_1$, enriched by the objects produced during the transformation (see Fig. 1). At the k-th composition step, the symbolic world is transformed by a service type s_k, which results in the symbolic world $\mathbf{w}_k$, representing the set of *final worlds* possible to obtain after k transformations of the initial worlds. The symbolic world w_k contains a number of "new" objects, produced in result of the subsequent transformations. If the consecutive transformations form a solution of the user query q, then among the "new" objects are these from out_q, requested by the user.

Following Def. 6 we have $W^q_{exp} = \left(in_q \cup inout_q \cup out_q, \mathcal{V}^q_{post}\right)$. First, we deal with the objects from $in_q \cup inout_q$, which are encoded directly over the symbolic world $\mathbf{w}_k$. Since these are the same objects as in the initial worlds, we know their indices, and therefore their states are encoded by the formula *ioExp*, defined as follows:

$$ioExp(\mathbf{w}_k, W^q_{exp}) = stF\Big(\mathbf{w}_k, \big(in_q \cup inout_q, \mathcal{V}^q_{post}(in_q \cup inout_q)\big)\Big),$$

where $\mathcal{V}^q_{post}\left(in_q \cup inout_q\right)$ is the family of the valuation functions $\mathcal{V}^q_{post}$ restricted to the objects from $in_q \cup inout_q$. Note that the formula encoding the types of the objects from $in_q \cup inout_q$ is redundant here. The

types are initially set by the formula encoding the initial worlds and the types are maintained between the consecutive worlds by the formulas encoding the subsequent world transformations (see Sec. 3.4).

Next, the objects of out_q need to be identified among the remaining objects of the symbolic world $\mathbf{w}_k$, i.e., among these represented by the symbolic objects of indices greater than $|\mathbf{w}_0|$. To this aim, we allocate a new symbolic world $\mathbf{w}_e$ with $e = k_{max} + 1$, containing all the objects from out_q. We encode their states by the formula $outExp$:

$$outExp(\mathbf{w}_e, W^q_{exp}) = stF\Big(\mathbf{w}_e, \big(out_q, \mathcal{V}^q_{post}(out_q)\big)\Big),$$

where $\mathcal{V}^q_{post}\big(out_q\big)$ is the family of the valuation functions $\mathcal{V}^q_{post}$ restricted to the objects from out_q.

Next, we need to encode the types of these objects. According to Def. 7, 9, and 13, a user query solution ends with a world (call it final) sub-compatible with an expected world. Notice that the objects from the final world matched to the objects from out_q, can be their subtypes. This is the reason for introducing the function $subT : \mathbb{O} \longmapsto 2^{\mathbb{N}} \setminus \emptyset$, which with every object o assigns the set of natural numbers corresponding to the type of o and all its subtypes.

Now, we define two formulas used for encoding objects compatibility. The first one encodes all subtypes of the objects from a given set O over a symbolic world $\mathbf{w}_i$:

$$sbF(\mathbf{w}_i, O) = \bigwedge_{o \in O} \bigvee_{t \in subT(o)} \mathbf{t}_{num(o),i} = t$$

The second formula encodes the compatibility of the attribute valuations of two symbolic objects:

$$eqF(\mathbf{o}_{i,j}, \mathbf{o}_{m,n}) = \bigwedge_{d=0}^{max_{at}} (\mathbf{a}_{i,d,j} = \mathbf{a}_{m,d,n}) \wedge (\mathbf{t}_{i,j} = \mathbf{t}_{m,n})$$

Finally, to complete the encoding of the expected worlds, we need a mapping between the objects from a final world $\mathbf{w}_k$ produced during the subsequent transformations and the objects from $\mathbf{w}_e$. To this aim we allocate p additional *mapping variables* in the symbolic world $\mathbf{w}_e$, where $p = |out_q|$. These variables, denoted by $\mathbf{m}_{0,e}, \ldots, \mathbf{m}_{p-1,e}$, are intended to store the indices of the objects from a final world, which are compatible with the objects encoded over $\mathbf{w}_e$. Thus, the last part of the expected worlds encoding is the formula:

$$mpF(\mathbf{w}_e, \mathbf{w}_k) = \bigwedge_{\mathbf{o}_{i,e} \in \mathbf{w}_e} \bigvee_{j=|\mathbf{w}_0|}^{|\mathbf{w}_k|-1} \big(eqF(\mathbf{o}_{i,e}, \mathbf{o}_{j,k}) \wedge \mathbf{m}_{i,e} = j\big)$$

Now, we can put all the components together and give the encoding of the expected worlds:

$$\mathcal{E}^q_k = ioExp(\mathbf{w}_k, W^q_{exp}) \wedge sbF\big(\mathbf{w}_e, out_q\big) \wedge outExp(\mathbf{w}_e, W^q_{exp}) \wedge mpF(\mathbf{w}_e, \mathbf{w}_k) \quad (3)$$

3.4 World Transformation

According to Def. 11, given a service type s, a world w, and a context function we can compute the world w' obtained after such a transformation. Now, we need to encode transformation sequences. In the previous subsection we presented the encoding of the initial and the expected worlds. Now, we need to allocate all intermediate symbolic worlds, and encode over them all the possible transformation sequences. Finally, the SMT-solver finds the valuations of such a formula, if there exists any, and they allow to discover the consecutive service types and the context functions. Therefore, in this subsection we show an idea how to build the formula encoding all the transformations $w \xrightarrow{s} w'$ over two subsequent symbolic worlds $\mathbf{w}$ and $\mathbf{w}'$.

For every planning step, i.e., for every transformation, we introduce additional sets of symbolic objects **in**, **io**, and **io**$'$, the integer variable **vs**, and two sets of integer mapping variables **pin** and **pio**. The symbolic objects from **in** and **io** are used to represent the input worlds of the service type being encoded. They correspond to the sets IN and IO of Def. 11. The variable **vs** represents the service type and the variables from **pin** and **pio** are used to encode the context functions. Finally, the objects from **io**$'$ are used to encode the objects modified during the transformation. The produced objects are encoded directly over the resulting symbolic world.

The transformation of the worlds represented by the symbolic world $\mathbf{w}_i$ via a service type s into a symbolic world $\mathbf{w}_{i+1}$ is defined as follows:

$$\mathcal{T}^s_i = inF(\mathbf{in}_i, \mathbf{io}_i, W^s_{pre}) \wedge cxF(\mathbf{w}_i, \mathbf{pin}_i, \mathbf{pio}_i) \wedge ouF(\mathbf{w}_{i+1}, \mathbf{io}'_i, W^s_{post}) \wedge \mathbf{vs}_i = num(s) \wedge cpF(\mathbf{w}_i, \mathbf{w}_{i+1}), \quad (4)$$

where inF and ouF stand for input and output worlds, resp., cxF encodes the context mappings, and cpF is responsible for "copying" symbolic objects of $\mathbf{w}_i$ not involved in the transformation to $\mathbf{w}_{i+1}$. The detailed definition of these formulas, built using similar constructions as for encoding a user query, is omitted due to the lack of space.

3.5 Multiset Blocking

The last component of our encoding are the blocking formulas, designed to prevent the search of solutions

Table 1: Experimental results.

(a) 1 plan in the search space

n	k	sat		unsat		time
		[s]	MB	[s]	MB	[s]
64	6	6.31	8.8	4.77	15.8	12.8
	9	19.49	17.5	38.08	83.8	58.7
	12	156.4	41.9	622.3	684	781
	15	469.7	229	> 2000		
128	6	7.29	10.8	6.20	18.7	14.8
	9	41.01	22.7	47.50	87.7	90.1
	12	203.2	38.5	1757	1917	1962
	15	382.1	47.9	> 2000		
256	6	16.66	17.9	8.55	22.8	27.1
	9	54.99	30.1	76.51	104	133
	12	315.4	49.0	1628	1559	1947
	15	1018	82.7	> 2000		

(b) 10 plans in the search space

first		next		unsat		time
[s]	MB	[s]	MB	[s]	MB	[s]
5.22	8.0	0.39	0.5	8.38	22.0	18.3
21.09	13.4	3.30	1.3	242.3	424	295
113.5	31.9	38.3	1.0	> 2000		
413	240	458	0.5	> 2000		
8.54	11.4	0.76	0.7	9.56	24.3	26.6
49.93	19.4	8.37	1.7	425.4	638	553
250.5	39.4	62.3	3.0	> 2000		
1850	78	?	?	> 2000		
11.93	16.1	0.64	0.6	18.34	36.6	38.1
113.3	32.5	5.54	2.0	810.8	1283	977
325.8	59.5	161	3.7	> 2000		
931	72.3	925	11	> 2000		

from already known classes (plans). To this aim, a convenient representation of an abstract plan is a multiset of the service types occurring in a user query solution.

In order to represent multisets, we need to encode counting of service types occurrences in transformation sequences. Let $\mathbb{B}^\star$ be the set of all sequences of Boolean values. We define a function $cnt : \mathbb{B}^\star \longmapsto \mathbb{N}$, which every Boolean sequence assigns the number of occurrences of the value **true**. The encoding of this function makes use of two abilities of modern SMT-solvers, namely *ite* (if-then-else) construct and the ability of defining internal functions. Thus, we encode the counting function as follows:

$$ct(b_1,\ldots,b_i) = \begin{cases} ite(b_i,1,0), & \text{for } i=1 \\ ite(b_i,1,0)+ct(b_1,\ldots,b_{i-1}), & \text{for } i>1 \end{cases}$$

where the expression $ite(b_i,1,0)$ returns 1 if b_i equals **true** and 0 otherwise. Now, having a user query solution we extract the sequence of service types $s = (s_1,\ldots,s_k)$ and we compute its multiset representation $M_s = \big((s_1,c_1),\ldots,(s_n,c_n)\big)$, where $s_i \in \mathbb{S}$, c_i is the number of occurrences of s_i in the sequence, and $1 \le i \le n \le k$. The formula blocking all solutions built over a single multiset is as follows:

$$bl(M_s) = \neg \bigwedge_{i=1}^{n} ct\big((\mathbf{vs}_1 = s_i),\ldots,(\mathbf{vs}_k = s_i)\big) = c_i$$

Assume that j abstract plans have been found, where each plan is represented by a multiset. The formula $\mathcal{B}_k^q$ blocking the solutions from all these plans is as follows:

$$\mathcal{B}_k^q = \bigwedge_{i=1..j}^{j} bl(M_{s_i}) \qquad (5)$$

4 EXPERIMENTAL RESULTS

Below we discuss the experimental results. We implemented our planner and evaluated its efficiency using the ontologies, the user queries, and the abstract plans generated by our Ontology Generator. The ontologies and the queries are generated randomly, but meeting the semantic rules, and in such a way that the number of the existing abstract plans is equal to a value of a given parameter.

The preliminary evaluation of our planner is presented in Tab. 1. The parameters of the experiments are: the number of existing abstract plans (1 and 10), the number of the service types in each ontology used (n), and the lengths of the abstract plans (k). The remaining table columns display the summary results of the experiments. All presented data are the average values, because each experiment was repeated from 5 to 10 times. The results include the following data: time and memory consumed by the solver while searching for the first abstract plan (*sat* and *first* columns), time and memory that the solver needs to find a different plan (*next*), time and memory used by the solver for checking that there are no different plans (*unsat*), and the total computation time (*time* column). The experiments have been performed on a computer equipped with 2.0GHz CPU and 8GB RAM, using the SMT-solver Z3 (de Moura and Bjørner, 2008). It follows from the experiments that the consumption of computing resources increases with the number of service types and the length of the plan. However, when the first plan is found, the next ones are computed several times faster.

In order to evaluate the efficiency of the encoding of the multiset blocking we have compared it with the sequence blocking. To this aim we performed

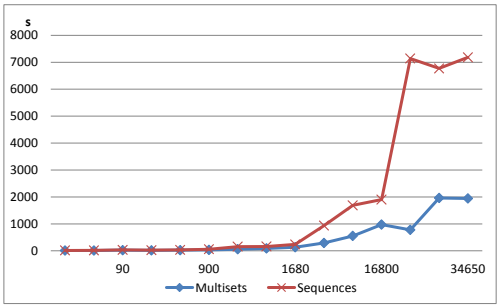

Figure 2: Multiset blocking versus sequence blocking.

additional experiments, using our encoding in which the formula (5) has been replaced by the one blocking the subsequent user query solutions. The results are summarised in Fig. 2. We used 15 benchmarks of Tab. 1, which do not exceed the 2000 sec. timeout. The values of the x-axis are the numbers of user query solutions, while the values of y-axis stand for the time needed to find all the plans. The general observation is that the more user query solutions exist, the more the multiset blocking outperforms the sequence blocking, and thus it works as we have expected. Moreover, during the 2000 sec. time limit, using sequences blocking we are able to generate all the solutions of length at most 9, while taking advantage of the multisets encoding we can find all the plans even for length 12. So, we are able to explore the search space 2^{24} times bigger in the same time.

We have compared the efficiency of our tool with another system. The paper (Nam et al., 2008) reports 7 experiments performed on a set of 413 *concrete* Web services, where SAT-time consumed for every composition varies from 40 to 60 sec. We have repeated these experiments translating the input data to the PlanICS ontology. PlanICS is able to find the shortest solution in just fractions of a second of SAT-time and in several seconds of the total computation time.

5 CONCLUSIONS

We presented an SMT-based approach to the abstract planning problem. Our main idea is to find significantly different abstract plans by partitioning the search space into equivalence classes of user query solutions. This concept has been realized by computing formulas, which encode multisets representing abstract plans, and blocking all solutions belonging to the plans already known. We have implemented our planner on the top of state of the art SMT-solver, and evaluated it using a number of scalable benchmarks. The experimental results are encouraging and confirm the efficiency of our approach.

REFERENCES

Ambroszkiewicz, S. (2004). Entish: A language for describing data processing in open distributed systems. *Fundam. Inform.*, 60(1-4):41–66.

Bell, M. (2008). *Introduction to Service-Oriented Modeling*. John Wiley & Sons.

Bentakouk, L., Poizat, P., and Zaidi, F. (2011). Checking the behavioral conformance of web services with symbolic testing and an SMT solver. In *Tests and Proofs*, volume 6706 of *LNCS*, pages 33–50. Springer.

Bersani, M. M., Cavallaro, L., Frigeri, A., Pradella, M., and Rossi, M. (2010). SMT-based verification of LTL specification with integer constraints and its application to runtime checking of service substitutability. In *SEFM*, pages 244–254.

de Moura, L. M. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag.

Doliwa, D., Horzelski, W., Jarocki, M., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M., and Zbrzezny, A. (2011). PlanICS - a web service compositon toolset. *Fundam. Inform.*, 112(1):47–71.

Elwakil, M., Yang, Z., Wang, L., and Chen, Q. (2010). Message race detection for web services by an SMT-based analysis. In *Proc. of the 7th Int. Conference on Autonomic and Trusted Computing*, ATC'10, pages 182–194. Springer.

Gehlot, V. and Edupuganti, K. (2009). Use of colored petri nets to model, analyze, and evaluate service composition and orchestration. In *System Sciences, 2009. HICSS '09.*, pages 1 –8.

Li, Z., O'Brien, L., Keung, J., and Xu, X. (2010). Effort-oriented classification matrix of web service composition. In *Proc. of the Fifth International Conference on Internet and Web Applications and Services*, pages 357–362.

Mitra, S., Kumar, R., and Basu, S. (2007). Automated choreographer synthesis for web services composition using I/O automata. In *ICWS*, pages 364–371.

Monakova, G., Kopp, O., Leymann, F., Moser, S., and Schäfers, K. (2009). Verifying business rules using an SMT solver for BPEL processes. In *BPSC*, pages 81–94.

Nam, W., Kil, H., and Lee, D. (2008). Type-aware web service composition using boolean satisfiability solver. In *Proc. of the CEC'08 and EEE'08*, pages 331–334.

Rao, J., Küngas, P., and Matskin, M. (2006). Composition of semantic web services using linear logic theorem proving. *Inf. Syst.*, 31(4):340–360.

Rao, J. and Su, X. (2004). A survey of automated web service composition methods. In *Proc. of SWSWPC'04*, volume 3387 of *LNCS*, pages 43–54. Springer.

Traverso, P. and Pistore, M. (2004). Automated composition of semantic web services into executable processes. In *The Semantic Web ISWC 2004*, volume 3298 of *LNCS*, pages 380–394.