Artur Niewiadomski
Wojciech Penczek
Agata Półrola

SMT-based Abstract Planning in PlanICS Ontology

Nr 1027

Warsaw, December 2012
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 and implementation, followed by preliminary experimental results.

Keywords: SMT, Web service composition, abstract planning

Streszczenie

Planowanie abstrakcyjne z wykorzystaniem SMT w ontologii Planics

Przedmiotem niniejszej pracy jest problem planowania abstrakcyjnego – pierwszy etap kompozycji usług sieciowych w systemie Planics. Prezentujemy rozwiązanie bazujące na kompaktowej reprezentacji planów abstrakcyjnych za pomocą wielozbiorów typów usług i redukcji problemu planowania do problemu spełnialności instancji SMT (Satisfiability Modulo Theories). Przedstawiamy zarówno teoretyczne aspekty planowania, jak również szczegóły symbolicznego kodowania i implementacji oraz wstępne wyniki eksperymentalne.

Słowa kluczowe: SMT, kompozycja usług sieciowych, planowanie abstrakcyjne
CONTENTS

Contents

1 Introduction 4
  1.1 Related Work .......................... 5

2 Abstract Planning 6
  2.1 Introduction to the PlanICS Ontology ............ 6
  2.2 Basic Definitions ........................... 7
  2.3 Abstract Planning Overview .................... 15
  2.4 World Transformations ......................... 16
  2.5 Plans .................................... 21

3 Symbolic Encoding and Implementation 24
  3.1 Introduction to SMT-LIB v2 language .............. 24
  3.2 Abstract Planning Algorithm .................... 26
  3.3 Symbolic Representation of Objects and Worlds .... 27
  3.4 User Query Encoding .......................... 27
  3.5 World Transformation Encoding .................. 31
  3.6 Multisets and Blocking Formulas ................. 33
  3.7 Implementation ............................. 34

4 Preliminary Experimental Results 35

5 Conclusions 38
1 Introduction

The main concept of Service-Oriented Architecture (SOA) [3] consists in using independent (software) components available via well-defined interfaces. Frequently, 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 well known as the Web Service Composition Problem (WSCP) [3, 1, 18]. In this paper, we follow the approach of our system PlanICS [9, 8], which has been inspired by [1].

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 dramatically 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. Moreover, we give some details of our symbolic encoding and implementation that are followed by preliminary
experimental results. To the best of our knowledge, the above approach is novel, and as our experiments show it is also very promising.

The theory is illustrated by a running example. The example corresponds to a situation in which a user would like to obtain a wooden arbour and to use the planner for finding services enabling him to reach the goal. We show a simple ontology and several abstract plans that allow to realize the task in different ways.

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 only when they can be composed automatically. The existing solutions to WSCP can be divided into several groups. According to [12] our approach belongs to AI planning methods, which include also approaches based on: automata theory [13], Petri nets [11], theorem proving [19], and model checking [20].

A composition method closest to ours is presented in [15], 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 take into account also the states of the objects, while [15] 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 [10], the authors of [4] take advantage of symbolic testing and execution techniques in order to check behavioural conformance of WS-BPEL specifications, [5] deals with a
service substitutability problem, while [14] 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

This section introduces Abstract Planning (AP) as the first stage of WSCP in the PlanICS framework. First, the PlanICS ontology is presented with a special focus on the features affecting the abstract planning process. Next, we provide some basic definitions and then explain the main goals of AP.

2.1 Introduction to the PlanICS Ontology

The OWL language [16] is used as the PlanICS ontology format. The concepts are organized in an inheritance tree of classes, all derived from the base class, called Thing. There are three descendants of Thing: Artifact, Stamp and Service (see Fig. 1).

The branch of classes rooted at Artifact is composed of the types of the objects, which the services operate on. Each object consists of a number of attributes, whereas an attribute consists of the name and the type. Note that the types of the attributes are irrelevant in the abstract planning phase as they are not used by the planner. The values of the attributes of an object determine its state, but in the abstract planning
it is enough to know only whether an attribute does have some value (i.e., is set), or it does not (i.e., is not set, so it is null).

The `Stamp` class and its descendants define special-purpose objects, often useful in constructing a user query, and in the planning process. A stamp is a specific type aimed at confirmation of the service execution. The specialized descendants of the `Service` class can produce the `stamp` being an instance of any subtype of `Stamp` and describing additional execution features (e.g., a price or an execution time). Note that each service produces exactly one confirmation object. The classes derived from `Artifact` and `Stamp` are called the `object types`.

Each class derived from `Service`, called a `service type`, stands for a description of a set of real-world services. It contains a formalized information about their activities. A service type affects a set of objects and transforms it into a new set of objects.

The detailed information about this transformation is contained in the attributes of a service type: the sets `in`, `inout`, and `out`, and the Boolean formulas `preCondition` and `postCondition` (pre and post for short). The mentioned sets enumerate the objects, which are processed by the service. The objects of the `in` set are read-only, i.e., they are passed unchanged to the world after. Each object of `inout` can be modified - the service can changes some values of its attributes. The objects of `out` are produced by the service. So, these are new objects, which enrich the world after.

### 2.2 Basic Definitions

AP makes intensive use of the service types and the object types defined in the ontology. As mentioned in the previous subsection, 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. We start from giving the basic definitions.

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 $\mathbb{A} \subseteq \mathbb{I}$. 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 $\mathbb{P}$.

Example 1 Consider the following exemplary ontology containing in addition to Thing also the class Artifact and Stamp. The class Artifact corresponds to the object type $(Artifact, \{id\})$ (the only attribute is an identifier) while the class Stamp corresponds to the object type $(Stamp, \{serviceType, serviceId, level\})$, introducing the attributes describing the service generating the stamp, and the position of this service in an execution sequence we consider.

We define also a transitive, irreflexive, and antisymmetric inheritance relation $Ext \subseteq \mathbb{P} \times \mathbb{P}$, such that $((t_1, A_1), (t_2, A_2)) \in Ext$ 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.

Example 2 Consider the object types depicted in Fig. 2:

- $(Ware, \{id, owner, location\})$
- $(Boards, \{id, owner, location, woodKind, thickness, volume\})$
- $(Nails, \{id, owner, location, size, weight\})$
- $(Arbour, \{id, owner, location, material, length, width, type\})$
- $(PriceStamp, \{serviceType, serviceId, level, price\})$
Figure 2: Object types inheritance in an example ontology and introducing new attributes by subtypes.

We have \((\text{Artifact}, \text{Ware}) \in \text{Ext}\) (i.e., Ware is a subclass of Artifact), as the set of attributes of Artifact is included in that of Ware. Similarly, \{(\text{Ware}, \text{Boards}), (\text{Ware}, \text{Nails}), (\text{Ware}, \text{Arbour}), (\text{Stamp}, \text{PriceStamp})\} \subseteq \text{Ext}. Moreover, from transitivity of \text{Ext} also \{\text{Artifact}\} \times \{\text{Boards}, \text{Nails}, \text{Arbour}\} \subseteq \text{Ext}.

An object \(o\) is a pair \(o = (id, type)\), where \(id \in I\) and \(type \in P\). That is, an object is a pair of the object name, and the object type, denoted respectively by \(id(o)\), and \(type(o)\), for a given object \(o\). The set of all objects is denoted by \(O\). Moreover, we define the function \(\text{attr} : O \rightarrow 2^A\) returning a set of the attributes for each object of \(O\).

**Example 3** Consider an object \(b = (\text{myBoards}, \text{Boards})\). We have \(id(b) = \text{myBoards}, type(b) = \text{Boards}, \text{and } attr(b) = \{id, \text{woodKind, thickness, volume, owner, location}\}.\)
**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 AP, 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:

$$
<\text{form}> ::= <\text{disj}>
$$

$$
<\text{disj}> ::= (<\text{disj}>) \mid <\text{conj}> \mid <\text{conj}> \text{ or } <\text{disj}>
$$

$$
<\text{conj}> ::= <\text{lit}> \mid <\text{conj}> \text{ and } <\text{lit}>
$$

$$
<\text{lit}> ::= \text{isSet}(o.a) \mid \text{isNull}(o.a) \mid \text{true} \mid \text{false}
$$

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

The above grammar defines a DNF formula without negations, i.e., a formula being the alternative of one or more clauses, referred to as *abstract clauses*. Every abstract clause is the conjunction of literals, specifying abstract values of object attributes using the functions $\text{isSet}$ and $\text{isNull}$. In the abstract formulas used in APP, we assume that no abstract clause contains both $\text{isSet}(o.a)$ and $\text{isNull}(o.a)$, for the same $o \in O$ and $a \in \text{attr}(o)$.

**Example 4** Let $O = \{b\}$ with $b = (\text{myBoards}, \text{Boards})$ as described before. An abstract formula over $o$ is for example $\psi := \text{isSet}(b.\text{owner}) \text{ and } \text{isNull}(b.\text{location}) \text{ or } \text{isSet}(b.\text{location}) \text{ and } \text{isSet}(b.\text{id})$ (which intuitively means that myBoards have either the owner set and the location not set, or the location and the identifier specified). The abstract clauses in the formula are: $\alpha_1 = \text{isSet}(b.\text{owner}) \text{ and } \text{isNull}(b.\text{location})$, and $\alpha_2 = \text{isSet}(b.\text{location}) \text{ and } \text{isSet}(b.\text{id})$.

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, the objects of \(inout_x\) can change their states, while \(out_x\) is supposed to contain only new objects.

Example 5 An example specification of a service type is as follows:

- \(in = \emptyset\),
- \(inout = \{(w, Ware)\}\),
- \(out = \{(stamp, PriceStamp)\}\),
- \(pre = isSet(w.owner)\),
- \(post = isSet(w.owner) \land isSet(w.id) \land isSet(stamp.price)\).

The specification corresponds to a Selling service type: it operates on a ware and produces a confirmation of its execution, i.e., a stamp containing the price (which is specified by inout and out respectively). To execute the service type it is required the owner of this ware to be known, and after the execution the (new) owner and the id of the ware sold are specified.

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 \(\varphi\) be an abstract formula over \(\emptyset\) s.t. \(\varphi = \bigvee_{i=1..n} \alpha_i\), where \(n \in \mathbb{N}\), and each \(\alpha_i\) is an abstract clause. A valuation of the object attributes over \(\alpha_i\) is the partial function \(v_{\alpha_i} : \bigcup_{o \in \emptyset} \{o\} \times attr(o) \rightarrow \{\text{true}, \text{false}\}\), where:
\[ v_{\alpha_i}(o,a) = \text{true} \text{ if } \text{isSet}(o.a) \text{ is a literal of } \alpha_i, \text{ or} \]

\[ v_{\alpha_i}(o,a) = \text{false} \text{ if } \text{isNull}(o.a) \text{ is a literal of } \alpha_i, \text{ or} \]

\[ v_{\alpha_i}(o,a) \text{ is undefined, otherwise.} \]

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

**Example 6** Consider the formula \( \psi \) from Example 4. We have:

\[ v_{\alpha_1}(b,\text{owner}) = \text{true}, \quad v_{\alpha_1}(b,\text{location}) = \text{false}, \quad v_{\alpha_2}(b,\text{location}) = \text{true} \]

and \( v_{\alpha_2}(b,\text{id}) = \text{true} \). In all the other cases the values of \( v_{\alpha_1} \) and \( v_{\alpha_2} \) are undefined.

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 \( A' \subseteq A \), \( f : A \rightarrow B \) be a total function, and \( f' : A \rightarrow B \) be a partial function, such that \( f' \) restricted to \( A' \) is total. We say that \( f \) is consistent with \( f' \), if both of the functions restricted to \( A' \) are equals, i.e., \( \forall a \in A' \ f'(a) = f(a) \).

Next, for a partial valuation function \( f \), by \( \text{total}(f) \) we denote the family of the total valuation functions that are consistent with \( f \). Note that for convenience we refer to the total valuation functions just as to valuation functions, but we explicitly state when a valuation function is partial.

Moreover, we define a family of the valuation functions \( \mathcal{V}_\varphi \) over the abstract formula \( \varphi \) as the union of the sets of the consistent valuation
functions over every abstract clause $\alpha_i$, i.e., $V_\varphi = \bigcup_{i=1}^n \text{total}(v_{\alpha_i})$. The restriction of the family of functions $V_\varphi$ to a set of objects $O$ and their attributes is defined as $V_\varphi(O) = \bigcup_{i=1}^n \text{total}(v_{\alpha_i}(O))$.

**Example 7** Consider the post formula from Example 5. Let $v_{\text{post}}$ be the partial valuation function over abstract clause post and let $\omega_0 = (w, \text{Ware})$ and $\text{st} = (\text{stamp}, \text{PriceStamp})$. We have:

$v_{\text{post}}(\omega_0, \text{owner}) = \text{true}$, $v_{\text{post}}(\omega_0, \text{id}) = \text{true}$, $v_{\text{post}}(\text{st}, \text{price}) = \text{true}$, and $v_{\text{post}}$ is undefined in all other cases.

Let $v = v_{\text{post}}(\omega_0)$, i.e., $v$ is the partial valuation function $v_{\text{post}}$ restricted to the singleton $\{\omega_0\}$. Now, we obtain: $v(\omega_0, \text{owner}) = \text{true}$, $v(\omega_0, \text{id}) = \text{true}$, and $v$ is undefined in all other cases. Moreover, total$(v) = \{v_1, v_2\}$, where

$v_1 = \{((\omega_0, \text{owner}), \text{true}), ((\omega_0, \text{id}), \text{true}), ((\omega_0, \text{location}), \text{true})\}$ and

$v_2 = \{((\omega_0, \text{owner}), \text{true}), ((\omega_0, \text{id}), \text{true}), ((\omega_0, \text{location}), \text{false})\}$.

We have also $V_{\text{post}}(\omega_0) = \{v_1, v_2\}$.

**Definition 5 (Worlds)** A world $w$ is a pair $(O_w, v(O_w))$, where $O_w \subseteq \emptyset$ 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 $V = \{v_1, \ldots, v_n\}$ is a family of total valuation functions and $O \subseteq \emptyset$ is a set of objects, then $(O, V(O))$ means the set $\{(O, v_i(O)) \mid 1 \leq i \leq n\}$, for $n \in \mathbb{N}$. Finally, the set of all worlds is denoted by $\mathbb{W}$.

Note that a valuation function used to define a world is always restricted to the set of the objects of the world, even if it is not stated explicitly.
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 $\text{spec}_x = (\text{in}_x, \text{inout}_x, \text{out}_x, \text{pre}_x, \text{post}_x)$ be a user query or a service type specification, where $x \in \{q, s\}$, resp. An interpretation of $\text{spec}_x$ is a pair of world sets $x = (W^x_{\text{pre}}, W^x_{\text{post}})$, defined as follows:

- $W^x_{\text{pre}} = (\text{in}_x \cup \text{inout}_x, \mathcal{V}^x_{\text{pre}})$, where $\mathcal{V}^x_{\text{pre}}$ is the family of the valuation functions over $\text{pre}_x$,

- $W^x_{\text{post}} = (\text{in}_x \cup \text{inout}_x \cup \text{out}_x, \mathcal{V}^x_{\text{post}})$, where $\mathcal{V}^x_{\text{post}}$ is the family of the valuation functions over $\text{post}_x$.

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

For a service type $(W^s_{\text{pre}}, W^s_{\text{post}})$, $W^s_{\text{pre}}$ is called the input world set, while $W^s_{\text{post}}$ - the output world set. The set of all the service types defined in the ontology is denoted by $S$. For a user query $(W^q_{\text{pre}}, W^q_{\text{post}})$, $W^q_{\text{pre}}$ is called the initial world set, while $W^q_{\text{post}}$ - the expected world set, and denoted by $W^q_{\text{init}}$ and $W^q_{\text{exp}}$, respectively. Notice that $\text{out}_x$ is supposed to contain only new objects, which are absent in $W^x_{\text{pre}}$, but present in $W^x_{\text{post}}$. In case of a service type $s$, the objects of $\text{out}_s$ are produced as the result of a world transformation to be defined in Sec. 2.4.

**Example 8** Consider the specification of a selling service presented in Example 5, where $\text{wo}$ and $\text{st}$ are defined as in Example 7. The suitable service type is $\text{Selling} = (W^\text{Selling}_{\text{pre}}, W^\text{Selling}_{\text{post}})$, with:

- $W^\text{Selling}_{\text{pre}} = (\{\text{wo}\}, \mathcal{V}^\text{Selling}_{\text{pre}})$, where $\mathcal{V}^\text{Selling}_{\text{pre}} = \text{total}(v)$,
  $v(\text{wo}, \text{owner}) = \text{true}$ and $v(\text{wo}, a)$ is undef. for $a \in \{\text{id}, \text{location}\}$;

- $W^\text{Selling}_{\text{post}} = (\{\text{wo}, \text{st}\}, \mathcal{V}^\text{Selling}_{\text{post}})$, where $\mathcal{V}^\text{Selling}_{\text{post}} = \text{total}(v')$ with $v'$ given by $v'(\text{wo}, \text{owner}) = \text{true}$, $v'(\text{wo}, \text{id}) = \text{true}$, $v'(\text{st}, \text{price}) = \text{true}$, and $v'$ being undefined for each other case.
2 Abstract Planning

2.3 Abstract Planning Overview

Overall, the main goal of AP 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. We start with the notions of compatibility of object states and worlds.

**Definition 7 (Compatible object states)** Let \( o, o' \in O \), 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., \( \text{type}(o) = \text{type}(o') \) or \( (\text{type}(o), \text{type}(o')) \in \text{Ext} \), and
- for all attributes of \( o \), we have that \( v' \) agrees with \( v \), i.e.,
  \[
  \forall a \in \text{attr}(o) \quad v'(o', a) = v(o, a). 
  \]

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.

**Example 9** Consider three objects \( o = (w, \text{Ware}) \), \( o' = (w', \text{Ware}) \), and \( o'' = (b, \text{Boards}) \), valuation functions \( v(o), v'(o') \), and partial valuation function \( v''(o'') \). Assume that:

- \( v(o, \text{location}) = v'(o', \text{location}) = v''(o'', \text{location}) \),
- \( v(o, \text{id}) = v'(o', \text{id}) = v''(o'', \text{id}) \),
- \( v(o, \text{owner}) = v'(o', \text{owner}) = v''(o'', \text{owner}) \),
then we have:
$v(o) \succ^{\text{obj}} v'(o'), \ v'(o') \succ^{\text{obj}} v(o), \ v''(o'') \succ^{\text{obj}} v(o), \text{ and } v''(o'') \succ^{\text{obj}} v'(o'), \text{ for } v'' \in \text{total}(v'').$

In order to identify similar worlds, we introduce the concept of worlds compatibility.

**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 $w' \succ^{\text{wrl}} w$, iff there exists a one-to-one mapping $\text{map} : O \rightarrow O'$ such that $\forall o \in O v'(\text{map}(o)) \succ^{\text{obj}} v(o)$.

Intuitively, the world $w'$ 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 $w'$. In our planning process we need also to identify similar worlds of different sizes. Thus, we introduce the notion of worlds sub-compatibility.

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

**Example 10** Consider the worlds $w = (\{o\}, v)$ and $w'' = (\{o''\}, v''_t)$, where $o, o'', v, v''_t$ are defined as in Example 9. It is easy to see that $w'' \succ^{\text{wrl}} w$. If we enrich the world $w''$ to contain any additional object (e.g., $o_1 = (A, \text{Arbour})$), obtaining this way a world $w_1$, i.e., $w_1 = (\{o'', o_1\}, v''_t)$, then it holds $w_1 \succ^{\text{swrl}} w$.

### 2.4 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 $\text{specs}_s$, 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 $ctx_s^{O'} : in_s \cup inout_s \cup out_s \mapsto O$ 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 $out_s$.

Note, that such a function could be defined if the number of objects in the set $O$ is at least the same as in the sum of sets defined by the service type $s$, i.e., $|O| \geq |in_s \cup inout_s \cup out_s|$.

**Example 11** Consider the service Selling presented in Example 8, and a set of objects $O' = \{a = (A, Arbour), n = (N, Nails), b = (B, Boards), st_1 = (s_1, PriceStamp), st_2 = (s_2, PriceStamp), st_3 = (s_3, Stamp)\}$. An example context function $ctx_{Selling}^{O'}$ can be given by:

$$ctx_{Selling}^{O'}((\text{stamp}, \text{PriceStamp})) = st_2, \ ctx_{Selling}^{O'}((w, \text{Ware})) = a.$$ 

Now, we can formally define a world transformation.

**Definition 11 (World transformation)** Let $w, w' \in \mathbb{W}$ be worlds, called a world before and a world after, resp., and $s = (W_{pre}^s, W_{post}^s)$ be a service type. Assume that $w = (O, v), w' = (O', v')$, where $O \subseteq O' \subseteq O$, 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 $OU = (O' \setminus O)$. 

Prace IPI PAN • ICS PAS Reports
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 $v_{pre}^s \in V_{pre}^s$ and $v_{post}^s \in V_{post}^s$, all the following conditions hold:

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

Intuitively, it means that:

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 $in_s$. The state of the objects from $IN$ is consistent with $pre_s$.

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 $inout_s$. The state of the objects from $IO$ is consistent with $pre_s$.

3. After the transformation the state of objects from $IO$ is consistent with $post_s$.

4. The world after contains a sub-world built over $OU$, which is compatible with a sub-world of some output world of the service type $s$, built over the objects from $out_s$. 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.
Example 12 Consider the service type Selling from Example 8, and the sets \( O = \{ a = (A, Arbour), n = (N, Nails), b = (B, Boards), st_1 = (s_1, PriceStamp), st_3 = (s_3, Stamp) \} \) and \( O' \) given as in Example 11. Moreover, let \( W = (O, total(v)) \) with \( v(a, owner) = \text{true} \), and \( v \) undefined otherwise. Let \( W' = (O', total(v')) \) with \( v'(a, id) = v'(a, location) = v'(a, owner) = v'(st_2, price) = \text{true} \) and \( v' \) undefined in all the other cases.

It is easy to see that Selling transforms the world \( w = (O, v_t) \) into the world \( w' = (O', v'_t) \) in the context \( \text{ctx}_{\text{Selling}}^{O'} \) given in the previous example, if \( v_t \in \text{total}(v) \), \( v'_t \in \text{total}(v') \), and none of the remaining objects does change during the transformation, i.e., \( v(\{n, b, st_1, st_3\}) = v'(\{n, b, st_1, st_3\}) \).

Now, let us define a sequence of world transformations.

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

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

Example 13 Let Select be a service type corresponding to selecting wares to be bought, given by the specification:
\[
\text{spec}_{\text{Select}} = (\text{in} = \emptyset, \text{inout} = \emptyset, \text{out} = \{(w, \text{Ware}), (\text{stamp}, \text{Stamp})\}, \text{pre} = \text{true}, \text{post} = \text{isSet}(w.\text{owner}) \text{ and isNull}(w.\text{id}))
\]
Moreover, let the service type Selling be defined as in the previous examples. Consider the world \( w_0 = (\{(N, Nails), (B, Boards), (s_1, PriceStamp)\}, v_0) \), for some total valuation function \( v_0 \), and the worlds \( w = (O, v_t), w' = (O', v'_t) \) from Example 12.
If we define a context function $ctx_O^\text{Select}$ by $ctx_O^\text{Select}((w,\text{Ware})) = (a,\text{Arbour})$ and $ctx_O^\text{Select}((\text{stamp},\text{Stamp})) = (s3,\text{Stamp})$, then we have that Select transforms $w_0$ into $w$ in the context $ctx_O^\text{Select}$. In Example 12 we have shown that Selling transforms $w$ into $w'$ in the context $ctx_O^{\text{Selling}}$. Thus, $\text{seq} = ((\text{Select},ctx_O^\text{Select}),(\text{Selling},ctx_O^{\text{Selling}}))$ is a transformation sequence.

Having the transformation sequences defined, we introduce the concept of user query solutions or simply solutions, needed to formulate the notion of a plan.

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

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 $q$.

**Example 14** Consider the following user query specification $\text{spec}_q$:

- $\text{in}_q = \{(N,\text{Nails}),(B,\text{Boards}),(s1,\text{PriceStamp})\}$,
- $\text{inout}_q = \emptyset$,
- $\text{pre}_q = \text{true}$,
- $\text{out}_q = \{(A,\text{Arbour})\}$,
- $\text{post}_q = \text{isSet}(A.owner)$.

One of the solutions of the user query $q$ is the transformation sequence $\text{seq} = ((\text{Select},ctx_O^\text{Select}),(\text{Selling},ctx_O^{\text{Selling}}))$ defined in Example 13.
2 Abstract Planning

2.5 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 $\text{count} : \bar{S} \times S \mapsto \mathbb{N}$ be such that $\text{count}(seq, s)$ 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 $\text{count}(seq, s) = \text{count}(seq', s)$ for each $s \in S$.

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.

**Example 15** Let us to extend our example ontology with two service types, given by the following specifications: $\text{spec}_{\text{Transport}} = (\text{in} = \emptyset, \text{inout} = \{(w, \text{Ware})\}, \text{out} = \{(\text{stamp}, \text{PriceStamp})\}$, $\text{pre} = \text{isSet}(w.\text{location})$ and $\text{isSet}(w.\text{id})$. 

---
post = isSet(w.location) and isSet(w.id) and isSet(stamp.price)), and

specWoodBuilding = (in = ∅, inout = {(b, Boards), (n, Nails)},
out = {(a, Arbour), (stamp, PriceStamp)},
pre = isSet(b.id) and isSet(n.id),
post = isSet(a.owner) and isSet(a.location) and isNull(b.id)
and isNull(n.id) and isSet(stamp.price)).

Transport service type is able to change location of some ware, while
WoodBuilding produces an arbour using boards and nails. Having available also service types Select and Selling, consider a user query q, given
by the specification:

spec_q = (in = ∅, inout = ∅, out = {(a, Arbour)}, pre = true,
post = isSet(a.owner) and isSet(a.location) and isSet(a.id)).
The shortest abstract plan is represented by the multiset M_1 = [Select,
Selling]. This plan is constituted by only one solution, similar to the one
from Example 13, where an arbour is selected and bought. Another plan
is represented by the multiset M_2 = [Select, Selling, Transport], where
the arbour is additionally delivered to the client. An example plan us-
ing the WoodBuilding service type is represented by the multiset M_3 =
[Select, Select, Selling, Selling, WoodBuilding], where boards and nails
needed to construct the arbour are first selected and bought. Note that
this plan consists of two solutions, that is, ignoring the contexts, the se-
quences (Select, Select, Selling, Selling, WoodBuilding) and (Select,
Selling, Select, Selling, WoodBuilding).

Finally, the following result shows that APP is a hard problem.

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

**Proof 1** Consider a 3-CNF formula \( \varphi \):

\[
\varphi = \varphi_1 \land \cdots \land \varphi_n
\]

where each clause \( \varphi_i = l_{i1}^i \lor l_{i2}^i \lor l_{i3}^i \) with \( l_{ij}^i = p \) or \( l_{ij}^i = \neg p \) for \( p \in PV \).
We build an ontology consisting of a set of service types $ST$ (to be defined later) and a set of object types $OT = \{O_i \mid 1 \leq i \leq n\} \cup \{O\}$, such that for any pair of the object types from $OT$ the inheritance relation does not hold, i.e., $\forall a,b \in OT (a,b) \notin Ext$. Moreover, the object type $O$ does have the attribute $p$ for each propositional variable $p$ in $PV(\varphi)$. Since we need only one instance of each object type, by $o$ we denote the instance of $O$, and by $o_i$ we mean an instance of an object type $O_i$.

Next, we build a user query $q$ such that there is an abstract plan over $ST$, which is a solution to $q$ iff $\varphi$ is satisfiable. The service types and the query are defined as follows:

- $ST = \bigcup_{i=1}^{n} \{s^i_1, s^i_2, s^i_3\}$,
- $\text{spec}_{s^i_j} =$ (\begin{align*}
in = \emptyset, \text{ inout} = \{o\}, \text{ out} = \{o_i\}, \\
\text{pre} = \text{post} = \begin{cases}
isSet(o.p) & \text{if } l^i_j = p, \\
isNull(o.p) & \text{if } l^i_j = \neg p
\end{cases}
\end{align*}$, for $1 \leq j \leq 3$,
- $\text{spec}_q =$ (\begin{align*}
in = \emptyset, \text{ inout} = \{o\}, \text{ out} = \{o_i \mid 1 \leq i \leq n\}, \\
\text{pre} = \text{post} = \text{true}
\end{align*})

Next, we show that each abstract plan satisfying $q$ over $ST$ defines the valuation, which satisfies $\varphi$. Let $\text{seq} = (s^1, \ldots, s^n)$ be a solution\footnote{ignoring context functions, for simplicity} of $q$, which transforms some world $w$ into $w'$. Notice that each $s^i$ is equal to some service type $s^i_j$ for $1 \leq j \leq 3$.

Define the valuation function $V: PV(\varphi) \rightarrow 2^{\{\text{true, false}\}}$ such that $V(p) = \text{true}$ iff $\text{isSet(o.p)}$ in $w'$ or $\neg(\text{isSet(o.p)}$ or $\text{isNull(o.p)}$ in $w')$. 
This means that either o.p has been set by some service type corresponding to the proposition p or o.p is undefined. The later reflects the situation when no service type of seq corresponds to p or ¬p. In this case p can be assigned either true or false, but we have decided that V(p) = true. It is easy to show that V(φ) = true.

Now, assume that V is a valuation such that V(φ) = true. We will show that there is an abstract plan, which is a user solution for q. Define s_i^j = s_i^j_j iff (l_i^j = p and V(p) = true) or (l_i^j = ¬p and V(p) = false) for some p ∈ PV(φ) and the minimal 1 ≤ j ≤ 3. This means that we take for s_i^j this service type s_i^j_j, which corresponds either to p in case V(p) = true or to ¬p in case V(p) = false, for which j is minimal.

It is easy to show that (s_1^1, ..., s_n^n) is a user query solution for q.

3 Symbolic Encoding and Implementation

This section presents a symbolic encoding of APP by an SMT formula, which is then tested for satisfiability by an SMT-solver, and sketches the implementation of our tool. First, we give a brief introduction to SMT-LIB v2 language, which is used to incorporate an SMT-solver into our planning engine. Next, 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 Introduction to SMT-LIB v2 language

The direct motivation for defining the SMT-LIB language has been the need of having a language common across the solvers in which one can express benchmark problems for the SMT-Competition event. The first version of the language was proposed in 2003 [17] by Ranise and Tinelli, but its successive revisions led to SMT-LIB version 2, which was announced in 2010 [2]. SMT-LIB v2 language is based around a set of commands interpreted by an SMT-solver. These commands change a
solver state or return properties of the solver state.

In order to encode the abstract planning problem introduced in the previous section, we make use of Boolean and integer variables, and we build a formula over them. Then, we ask a solver whether the formula is satisfiable, i.e., is there such an interpretation of the variables that evaluates the whole formula to \textit{true}.

Listing 1 presents a simple example introducing several basic SMT-LIB v2 commands, which aim is to check satisfiability of the following expression: \((x - y > 2) \land (p \land \neg q)\).

Listing 1: SMT-LIB v2 example

; declaration of integer variables
(declare-fun x () Int)
(declare-fun y () Int)

; declaration of propositional variables
(declare-fun p () Bool)
(declare-fun q () Bool)

; constructing a formula and putting it as assertion
(assert (and (> (- x y) 2) (and p (not q))))

; checking satisfiability
(check-sat)

; if satisfiable, show the valuation
(get-model)

; ; ; result returned by the solver

sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    3)
  (define-fun q () Bool)
false)
(define-fun p () Bool
   true)
)

A comprehensive tutorial on SMT-LIB v2 can be found in [6].

3.2 Abstract Planning Algorithm

The core of our SMT-based abstract planner is an adaptation of a sym-
bolic Bounded Model Checking (BMC) method. The main idea of BMC
is to search for an execution of a system of some length $k$, which satisfies
some property.

In the case of APP, given an ontology, a user query, and some addi-
tional parameters $k_{min}$ and $k_{max}$, the planner is searching for user query
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 ab-
stract plan. As a result, a blocking formula is computed, which is used
to exclude from a further search all the solutions belonging to this ab-
stract 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 $\varphi_k^q$:

$$\varphi_k^q = \mathcal{I}_q \bigwedge_{i=1..k} \bigvee_{s \in S} \mathcal{T}_{i}^s \land \mathcal{E}_k^q \land \mathcal{B}_k^q,$$

where $\mathcal{I}_q$ and $\mathcal{E}_k^q$ are formulas encoding the initial and the expected
worlds, respectively, $\mathcal{T}_{i}^s$ encodes a transformation of one world into an-
other by a service type $s$, and $\mathcal{B}_k^q$ stands for a blocking formula.
Theorem 2 (Correctness of encoding) The formula $\varphi_k^q$ is satisfiable iff there is a user query $q$ solution of length $k$.

Proof 2 By sketch of the encoding of APP, represented by the formula $\varphi_k^q$, provided in the next subsections.

3.3 Symbolic Representation of 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 $\text{num}: \mathbb{A} \cup \mathbb{P} \cup \mathbb{S} \cup \mathbb{O} \rightarrow \mathbb{N}$, 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: $o_{i,j} = (t_{i,j}, a_{i,0,j}, a_{i,1,j}, ..., a_{i,max_{at}-1,j})$, where $t_{i,j}$ is the type variable, $a_{i,x,j}$ 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 $w_j$, while the number of the symbolic objects in $w_j$ - by $|w_j|$. Fig. 3 shows subsequent symbolic worlds of a transformation sequence.

3.4 User Query Encoding

In order to encode the set $W_{\text{init}}^q$ by a symbolic world $w_0$, we allocate the variables needed to represent the objects from $in_q \cup inout_q$. Then, we build the formula $I_q^g$ over these variables, which encodes the types and
the states of the objects from the initial worlds:

\[ \mathcal{I}^q = tpF(w_0, \text{in}_q \cup \text{inout}_q) \land stF(w_0, W^q_{\text{init}}) \]  

(2)

The formula \( tpF(w_i, O) \) encoding the types of the objects \( O \) over some symbolic world \( w_i \), is defined as:

\[ tpF(w_i, O) = \bigwedge_{o \in O} t_{\text{num}(o),i} = \text{num}(\text{type}(o)) \]

The formula \( stF(w_i, W) \) encodes the states of the objects from the worlds \( W = (O, V) \) over the symbolic world \( w_i \):

\[ stF(w_i, W) = \bigvee_{v \in V} \bigwedge_{o \in O} \bigwedge_{a \in \text{attr}(o)} vF(w_i, v, o, a), \]

where \( vF(w_i, v, o, a) \) is the expression encoding the valuation \( v \) of the attribute \( o.a \) over the variables of the symbolic world \( w_i \), defined as
3 Symbolic Encoding and Implementation

follows:

\[ vF(w_i, v, o, a) = \begin{cases} a_{num(o),num(a),i}, & \text{if } v(o, a) = \text{true}, \\ \neg a_{num(o),num(a),i}, & \text{if } v(o, a) = \text{false}, \\ \text{true}, & \text{if } v(o, a) \text{ is undef}. \end{cases} \]

Thus, the symbolic world \( w_0 \) represents the initial worlds. Then, after the first transformation we obtain the symbolic world \( w_1 \), enriched by the objects produced during the transformation (see Fig. 3). At the \( k \)-th composition step, the symbolic world is transformed by a service type \( s_k \), which results in the symbolic world \( w_k \), 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 \( W^q_{exp} = (in_q \cup inout_q \cup out_q, V^q_{post}) \). First, we deal with the objects from \( in_q \cup inout_q \), which are encoded directly over the symbolic world \( w_k \). 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:

\[ ioExp(w_k, W^q_{exp}) = stF(w_k, (in_q \cup inout_q, V^q_{post}(in_q \cup inout_q))), \]

where \( V^q_{post}(in_q \cup inout_q) \) is the family of the valuation functions \( V^q_{post} \) 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. 3.5).

Next, the objects of \( out_q \) need to be identified among the remaining objects of the symbolic world \( w_k \), i.e., among these represented by the symbolic objects of indices greater than \( |w_0| \). To this aim, we allocate
a new symbolic world \( w_e \) with \( e = k_{\text{max}} + 1 \), containing all the objects from \( \text{out}_q \). We encode their states by the formula \( \text{outExp} \):

\[
\text{outExp}(w_e, W_{\text{exp}}^q) = stF\left( w_e, (\text{out}_q, \mathcal{V}_{\text{post}}^q(\text{out}_q)) \right),
\]

where \( \mathcal{V}_{\text{post}}^q(\text{out}_q) \) is the family of the valuation functions \( \mathcal{V}_{\text{post}}^q \) restricted to the objects from \( \text{out}_q \).

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 \( \text{out}_q \), can be their subtypes. This is the reason for introducing the function \( \text{subT} : \emptyset \mapsto 2^\mathbb{N} \setminus \emptyset \), 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 \( w_i \):

\[
\text{sbF}(w_i, O) = \bigwedge_{o \in O} \bigvee_{t \in \text{subT}(o)} t_{\text{num}(o),i} = t
\]

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

\[
\text{eqF}(o_{i,j}, o_{m,n}) = \bigwedge_{d=0}^{\text{max}_{\text{at}}} (a_{i,d,j} = a_{m,d,n}) \land (t_{i,j} = t_{m,n})
\]

Finally, to complete the encoding of the expected worlds, we need a mapping between the objects from a final world \( w_k \) produced during the subsequent transformations and the objects from \( w_e \). To this aim we allocate \( p \) additional \textit{mapping variables} in the symbolic world \( w_e \), where \( p = |\text{out}_q| \). These variables, denoted by \( m_0,e, \ldots, m_{p-1,e} \), are intended to store the indices of the objects from a final world, which are
compatible with the objects encoded over $w_e$. Thus, the last part of the expected worlds encoding is the formula:

$$mpF(w_e, w_k) = \bigwedge_{o_{i,e} \in w_e} \bigvee_{j = |w_0|} (eqF(o_{i,e}, o_{j,k}) \land m_{i,e} = j)$$

(3)

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

$$\mathcal{E}^q_k = ioExp(w_k, W^q_{exp}) \land sbF(w_e, out_q) \land outExp(w_e, W^q_{exp}) \land mpF(w_e, w_k)$$

(4)

### 3.5 World Transformation Encoding

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 $w$ and $w'$.

For every planning step, i.e., for every transformation, we introduce a transformation template, i.e., additional sets of symbolic objects $in$, $io$, and $io'$, the integer variable $vs$, and two sets of integer mapping variables $pin$ and $pio$ (see Fig. 4). 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 $w_i$ via a service type $s$ into a symbolic world $w_{i+1}$ is defined as follows:

$$
\mathcal{T}_i^s = ouF(w_{i+1}, io_i', W_{post}^s) \land inF(in_i, io_i, W_{pre}^s) \land \\
\text{cx}F(w_i, pin_i, pio_i) \land \text{cp}F(w_i, w_{i+1}) \land \text{vs}_i = \text{num}(s),
$$

where $inF$ encodes input worlds:

$$
inF(in_i, io_i, W_{pre}^s) = stF\left(in_i, (in_s, \gamma_{pre}(in_s))\right) \land \\
stF\left(io_i, (inout_s, \gamma_{pre}(inout_s))\right),$$
ouF stands for output worlds:

\[
\text{ouF}(\mathbf{w}_{i+1}, \mathbf{i}o'_i, W^s_{\text{post}}) = \text{stF}(\mathbf{i}o'_i, (\text{inout}_s, \mathcal{V}^s_{\text{post}}(\text{inout}_s))) \land \\
\text{stF}(\mathbf{w}_{i+1}, (\text{out}_s, \mathcal{V}^s_{\text{post}}(\text{out}_s))),
\]

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 is omitted here, since they are built of similar constructions as Formula (3), used for encoding a user query.

### 3.6 Multisets and Blocking Formulas

The last component of our encoding are the blocking formulas, designed to prevent the search of solutions 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 \( \mathbb{B}^* \) be the set of all sequences of Boolean values. We define a function \( cnt : \mathbb{B}^* \to \mathbb{N} \), 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 \( \text{ite} \) (if-then-else) construct and the ability of defining internal functions. Thus, we encode the counting function as follows:

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

where the expression \( \text{ite}(b_i, 1, 0) \) 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

\[
M_s = ((s_1, c_1), \ldots, (s_n, c_n)),
\]

where \( s_i \in \mathbb{S}, c_i \) is the number of occurrences of \( s_i \) in the sequence, and \( 1 \leq i \leq n \leq k \). The formula blocking
all solutions built over a single multiset is as follows:

\[
\text{block}(M_s) = - \bigwedge_{i=1}^{n} ct\left((vs_1 = s_i), \ldots, (vs_k = s_i)\right) = c_i
\]

Assume that \( j \) abstract plans have been found, where each plan is represented by a multiset. The formula \( B^q_k \) blocking the solutions from all these plans is as follows:

\[
B^q_k = \bigwedge_{i=1..j} \text{block}(M_{s_i}) \quad (6)
\]

### 3.7 Implementation

The main components of our tool are depicted in Fig. 5. The ontology and the user query are first processed by the PlanICS language parser, and then by **Ontology Wrapper**, which is responsible for preparing all the auxiliary structures and mappings to be used by **Incremental Planner** (IP, for short). IP and SMT-solver perform the planning process by
Table 1: Experimental results: 1 plan in the search space

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
<th>sat</th>
<th>unsat</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[s]</td>
<td>MB</td>
<td>[s]</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>6.31</td>
<td>8.8</td>
<td>4.77</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>19.49</td>
<td>17.5</td>
<td>38.08</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>156.4</td>
<td>41.9</td>
<td>622.3</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>469.7</td>
<td>229</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td>128</td>
<td>6</td>
<td>7.29</td>
<td>10.8</td>
<td>6.20</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>41.01</td>
<td>22.7</td>
<td>47.50</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>203.2</td>
<td>38.5</td>
<td>1757</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>382.1</td>
<td>47.9</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td>256</td>
<td>6</td>
<td>16.66</td>
<td>17.9</td>
<td>8.55</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>54.99</td>
<td>30.1</td>
<td>76.51</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>315.4</td>
<td>49.0</td>
<td>1628</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>1018</td>
<td>82.7</td>
<td>&gt; 2000</td>
</tr>
</tbody>
</table>

searching for subsequent solutions on increasingly greater depths (see Sec. 3.2). The solver works in an incremental mode, i.e., the variables and some learned information are reused during the subsequent planning steps. The planner keeps an instance of the solver running in the interactive mode and the communication between them is implemented using our *ArtSMTLib* library and SMT-LIB v2 commands. This solution enables the cooperation with any SMT-LIB v2 compatible solver supporting the interactive mode, which comes with a minimal implementation effort only.

4 Preliminary Experimental Results

In this section we discuss the preliminary 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 in a
The preliminary evaluation of our planner is presented in Tab. 1 and Tab. 2. 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 \((sat \text{ and } \textit{first} \text{ columns})\), 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 \((time \text{ column})\), including loading of the ontology, encoding of the planning problem, time consumed by the solver, and the analysis of the solutions. The experiments have been performed

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
<th>first</th>
<th>next</th>
<th>unsat</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[s]</td>
<td>[s]</td>
<td>[s]</td>
<td>[s]</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>5.22</td>
<td>8.0</td>
<td>0.39</td>
<td>8.38</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>21.09</td>
<td>13.4</td>
<td>3.30</td>
<td>242.3</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>113.5</td>
<td>31.9</td>
<td>38.3</td>
<td>458</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>413</td>
<td>240</td>
<td>458</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td>128</td>
<td>6</td>
<td>8.54</td>
<td>11.4</td>
<td>0.76</td>
<td>9.56</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>49.93</td>
<td>19.4</td>
<td>8.37</td>
<td>425.4</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>250.5</td>
<td>39.4</td>
<td>62.3</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>1850</td>
<td>78</td>
<td>?</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td>256</td>
<td>6</td>
<td>11.93</td>
<td>16.1</td>
<td>0.64</td>
<td>18.34</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>113.3</td>
<td>32.5</td>
<td>5.54</td>
<td>810.8</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>325.8</td>
<td>59.5</td>
<td>161</td>
<td>&gt; 2000</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>931</td>
<td>72.3</td>
<td>925</td>
<td>&gt; 2000</td>
</tr>
</tbody>
</table>
4 Preliminary Experimental Results

Figure 6: Multiset blocking versus sequence blocking

The general observations following from the experiments can be summarised as follows. 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 additional experiments, using our encoding in which the formula (6) has been replaced by the one blocking the subsequent user query solutions. The results are summarised in Fig. 6. We used 15 benchmarks of Tab. 1 and Tab. 2, which do not exceed the 2000 sec. time-out. 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. This means that we are able to explore the search space \(256^3 = 2^{24}\) times bigger in the same 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. A significant progress, in comparison to the previous version of PlanICS, has been made as before it was not possible to find all abstract plans.

**Acknowledgements.** The research described in this paper has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.
REFERENCES

References


REFERENCES


Pracę zgłosił Antoni Mazurkiewicz.

Adres autora: Artur Niewiadomski  
Instytut Informatyki UPH  
ul. 3-Maja 54  
08-110 Siedlce, Polska

Wojciech Penczek  
Instytut Podstaw Informatyki PAN  
ul.Jana Kazimierza 5  
01-248 Warszawa, Polska

Agata Półrola  
Wydział Matematyki i Informatyki UŁ  
ul. Banacha 22  
90-238 Łódź, Polska

E-mail: artur@ii.uph.edu.pl  
penczek@ipipan.waw.pl  
polrola@math.uni.lodz.pl

Symbol klasyfikacji rzeczowej: II.C.2.4

Na prawach rękopisu
Printed as manuscript

Nakład 90 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w grudniu 2012 r. Wydawnictwo IPI PAN. ISSN 0138-0648.