Combining Ontology Reductions with New Approaches to Automated Abstract Planning of PlanICS☆

Artur Niewiadomski[a,*], Wojciech Penczek[a,b,*], Jaroslaw Skaruz[a], Maciej Szreter[b], Agata Półrola[c]

[a] *Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, 3-Maja 54, 08-110 Siedlce, Poland*
[b] *Institute of Computer Science, Polish Academy of Sciences, Jana Kazimierza 5, 01-248 Warsaw, Poland*
[c] *Faculty of Mathematics and Computer Science, Lodz University, Banacha 22, 90-238 Lodz, Poland*

Abstract

The paper deals with the first phase of Web Service Composition – the abstract planning problem, performed by the tool Planics. We discuss three algorithms based on multisets of service types being a concise representation of abstract plans. The first algorithm reduces the abstract planning problem to a satisfiability problem for an SMT-solver (SMT stands for the Satisfiablity Modulo Theories), the second one follows a GA-based approach (GA stands for a genetic algorithm), while the third hybrid algorithm combines the two above. All the algorithms are applied to ontologies, which are efficiently reduced using a graph database approach. The paper presents theoretical aspects of the framework together with many examples, and the extensive experimental results of the four algorithms followed by their analysis and comparison with each other and other approaches.

Keywords: Tool PlanICS, Service-Oriented Architecture, Web Service Composition, Genetic Algorithms, Satisfiability Modulo Theories, Abstract Planning, Hybrid algorithms, Graph Databases,

1. Introduction

The main concept of Service-Oriented Architecture (SOA) [1] consists in using independent (software) components that are easily available via interfaces. Typically, Web services have to be composed in order to satisfy the user's goal as a simple web service does not need to satisfy it. In order to relieve the user of manually defining plans, selecting services and their providers, many algorithms solving the Web Service Composition Problem (WSCP) [1, 2, 3, 4] have been recently developed [5, 6, 7, 8]. Since WSCP is hard but a very important problem, it attracts a lot of attention in the Web service community. In this paper we deal with WSCP and to this aim we follow our earlier results on the system Planics [9, 10].

It is assumed that one can construct a hierarchy of *classes*, organised in an *ontology*, containing all underlined web services and the objects processed by them. Moreover, the planning process is divided into several stages in order to reduce the task complexity by restricting the number of concrete services to be considered. In the first phase the planner operates on *classes of services* corresponding to sets of real-world services of common functionalities, matching them into an *abstract plan* [11, 12]. The second stage deals with *concrete services*, refining the abstract plan into a *concrete one* [13].

In this paper our focus is on the abstract planning problem only, attempting to find all *significantly different* plans. While the existing approaches (see Sect. 1.1) are quite efficient for ontologies of medium size, their computational time becomes unacceptable for ontologies of larger sizes. Therefore, our solution consists in combining a symbolic method based on applications of SMT-solvers [14] and genetic algorithms [15, 16], to ontologies reduced with graph database methods.

The paper is organized in the following way. Firstly, the abstract planning problem (APP, for short) is defined and shown to be NP-hard. Then, our original approach to APP using multisets of service types as a concise representation of abstract plans is presented. Note that a multiset representation can be even exponentially more compact than the set of all its linearisations. This approach is combined with delegating a task for an SMT-solver and a genetic algorithm, which in addition to a database ontology reduction method are the main contributions of this paper.

In the SMT-based approach, the blocking formulas are used for pruning the search space with different 'interleavings' of linearisations of the already generated plans. In the GA-based approach, an individual is constructed for a multiset of service types on which all GA operations are executed. This gives a great improvement in comparison to a linear representation of an individual as the correct order of the service types can be neglected. A linearization of an individual is obtained for computing its fitness value only. The algorithm stores each abstract plan newly found. In the subsequent iterations all individuals similar to the abstract plans stored are 'punished' by decreasing their fitness value. Note that for the search space pruning (by means of blocking the similar solutions) the order of service types does not matter because every sequence is projected to a multiset. However, it is not known whether a multiset con-

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

*Corresponding author
Email address: artur.niewiadomski@uph.edu.pl (Artur Niewiadomski)

stitutes a plan before a valid sequence is found. Moreover, in addition to a multiset, a valid sequence is needed in the next composition stages when the ordering is definitively important (e.g., while concrete services are to be called).

Our extensive experiments show that the above two methods are complementary. While the SMT-based approach is precise but suffers from a long computation time, the GA-based approach is much faster, but the probability of finding a solution decreases if the lengths of plans grow. Therefore, we present also the third hybrid approach which shares the advantages of both the planners. In the hybrid algorithm each iteration of GA is followed by the SMT-based procedure which modifies several individuals meeting required conditions in order to obtain solutions of APP. In following sections, details of the algorithms are given while experimental results are presented in Section 7.

In real-world applications, an ontology is usually very large while user queries tend to be local, producing plans containing small numbers of services and objects, compared to the overall ontology size. The Planics planners developed so far work on whole ontologies what hinders their performance and leaves a lot of space for improvements. While the idea of pruning ontologies was examined before (see [17]), our paper presents a novel and original approach of using a graph database to pruning ontologies on the basis of user queries. Our experimental results confirm that this approach can result in a very significant improvement in the computing time. Moreover, our experiments show that the ontology pruning can greatly improve also the performance of an external tool (Fast Downward [18]) adapted to solve APP due to suitable translations. In some cases the ontology reduction makes that tool efficient, even when the size of the original problem is prohibitive for them.

The remainder of the paper is organised as follows. Section 1.1 discusses related work. Abstract planning problem is dealt with in Section 2. SMT-based approach to APP is given in Section 3 while Section 4 shows how GA can be applied to this aim. The hybrid solution is described in Section 5. Section 6 shows how a graph database can prune an ontology, guided by the user query. Section 7 presents the experimental results of our planning tool. The last section concludes the results.

Additionally, we provide a supplementary material online [19] containing more examples illustrating the theory, the proofs of the two theorems, and other resources which may be of interest to the reader.

1.1. Related Work

Some preliminary results describing our SMT- and GA-based abstract planning methods have been published in conference proceedings: an SMT-based method in [11], a GA-based approach in [20, 12], and a hybrid solution in [21]. This paper extends and refines the above approaches, presents them all together with examples, and, for many benchmarks, compares their efficiency also with two other tools.

The existing solutions to WSCP, belonging to AI planning methods, can be divided [5] into numerous approaches based on: automata theory [22], situation calculus [23], Petri nets [24], theorem proving [25], and model checking [26, 27, 28]. In [6] a similar SAT-based approach to solve WSCP is presented, but the states of the objects are not considered and plans are not represented by multisets. Other applications of SMT to WSCP deal with automatic verification and testing e.g., a message race detection problem is considered in [29], a behavioural conformance of WS-BPEL specifications is checked in [30], a service substitutability problem is investigated in [31], and WS-BPEL specifications are verified against business rules in [32]. In [33] a similar GA-based approach is presented, but the algorithm is used to one phase planning (abstract combined with the concrete one). Individuals represented by multisets were used in [34], but without computing linearizations of them to find the fitness value.

There are several papers solving the web composition problem by modeling it in the domain of graphs, with solutions corresponding to reachability in graphs. In these papers, services are based on the IOPR (input, output, precondition, result) paradigm, similarly to our approach. However, none of the papers distinguishes between abstract and concrete planning in our sense, and neither uses graph databases. To the best of our knowledge, we could also find no experimental analysis showing that graph-based approaches can effectively deal with ontologies with significant numbers of services and objects. In [35] information about inputs and outputs of services is represented by interface automata. In dependency graphs, the vertices correspond to object types while the edges to services processing them. There is no experimental evaluation. Both services and objects processed by them are represented by graph vertices [36], and backward chaining is used for searching plans satisfying the user query. The paper does not offer a complete solution. In [37], the vertices correspond to web services, but there is no abstract phase and no objects directly represented in the graph. Edges are added at the basis of concrete values of the parameters. There are no experimental results and the authors seem not to care much about the efficiency of the method. Weighted graphs are used [38] for modeling parameters such as cost, execution time and availability, and presents respective graph algorithms for solving the corresponding composition problem. A graph approach to composing web services is described in [17]. The way of modeling is similar to ours, with input and ouput parameters, but without the formalized ontology. Contrary to our approach the algorithm does not reduce the problem to testing reachability in graphs, but runs algorithms searching and updating the graphs on-the-fly. No experiments are presented showing how the approach scales for increased numbers of services. It does not use a graph database, and does not deal with the abstract planning of any kind.

Another graph-based approach was examined in [39], where the concepts (being inputs and outputs of services) are defined in an ontology using an inheritance relation, but they do not have any internal structure. The paper also does not test the reachability, but constructs a graph by an exhaustive fix-point search with testing semantic conditions on-the-fly. The result of the graph search is a possibly non-optimal concrete plan that is later improved which corresponds to the concrete planning phase of Planics. Some results reporting a performance evaluation for different numbers of services are, given, but the graph

search is not distinguished from the optimization stages. In [40] ontologies are annotated with semantics information, and a graph algorithm is used with similarity measures describing how services are matched.

The key difference between the papers described above and our approach is that we model matching of the service input and output at the graph level, by reducing it to the problem of testing reachability in graphs. Graph nodes model not only services, but also objects consumed and produced by services. The abstract planning allows to reduce matching of service and object types to the problem of finding paths between selected graph nodes. The other approaches use graphs as a model for representing state spaces, but test matching objects to services by calling specialized algorithms what hampers the performance. Another original feature of our method consists in using graph databases for representing graphs and performing operations on them.

Graph databases [41, 42] are a relatively recent addition in the domain of the NoSQL databases, defined by rejecting the traditional database model of relational tables, and using alternative solutions, in this case graphs. Neo4j is one of the most popular implementations. In [43] web service composition using the MapReduce approach aimed at processing Big Data is considered. Our method uses a different solution, but shares the idea of exploiting efficient tools developed for dealing with big amounts of data.

Recently, several papers have explored the connection between multi-agent systems (MAS) and composition of web services (WSs), where agents can provide WSs, use WSs for communication, and compose WSs. For example, [44] describes the model-driven development (MDD) of MAS communicating over semantic web in order to exchange and evaluate semantic information. In [45] the framework of [44] is extended by introducing a domain-specific modeling language SEA_ML, which uses the metamodel independent of the agent platform. The Planics approach enables to apply the MAS techniques described above. For example, the definition of execution semantics would allow for an application of formal tools reasoning about knowledge in multi-agent systems, in the way similar to [46] uses the formal semantics of SEA_ML for performing formal validation. Epistemic temporal logic [47] could be used for getting information from web services or composing them.

In the next section, after introducing the Planics ontology, we continue the discussion by comparing Planics with approaches to WSCP adapting WSMO [48] and OWL-S [49] concepts.

2. Abstract Planning Phase

We start with introducing APP of Planics. We present the Planics ontology, focusing mainly on the features affecting the abstract planning process and then basic definitions, which enable us to explain what abstract planning is about.

2.1. Planics Ontology

The Planics ontology format exploits the OWL language [50], where the concepts build an inheritance tree of *classes*. All the classes in an ontology are derived from one base class called *Thing*; the class has three direct descendants of the names *Artifact*, *Stamp*, and *Service*. The further descendants are "domain-dependent", i.e., they can differ depending on the area covered by the given ontology. However, the contents of each ontology should meet the rules presented below. A fragment of an example ontology is presented in Fig. 1.

By *object types* we mean the classes derived from *Artifact* and *Stamp*. The branch of classes rooted at *Artifact* aims at storing the types of objects the services operate on. Each object is composed of a number of attributes, whereas each attribute definition consists of the name and the type. The values of the attributes determine the current state of the object. However, in the abstract planning phase the types of the object attributes are irrelevant (i.e., they are not used by the planner). Similarly, the planner does not consider the exact values of the attributes, focusing only on the fact whether an attribute has some value (i.e., is set) or not (i.e., is not set, so it is null).

The *Stamp* class and its descendants define special-purpose object types, aimed at confirmation of the service execution. Moreover, subclasses of *Stamp* can describe additional service execution features, like a price or an execution time. Each service, when executed, produces exactly one stamp - a confirmation object. The stamps are useful in constructing a user query, as well as in the planning process.

The classes from the branch rooted at *Service*, called *service types*, correspond to sets of real-world services. Each set is featured by a common activity, a formalised description of which is provided by the (values of) class attributes. This activity affects a set of objects and transforms it into a new set of objects. The attributes used to describe the transformation are as follows: the sets *in*, *inout*, and *out* enumerating the objects processed by the service, and the Boolean formulas *pre* and *post* specifying the conditions satisfied by these objects before and after the service execution. The sets distinguish between the objects which are read-only, i.e., required for the service to be executed and passed unchanged to the set of objects resulting from this execution (the set *in*), these which values of attributes can be modified by the service (the set *inout*), and these which are produced by the service (the set *out*).

Let us also make a comment on the inheritance in the Planics ontologies. Since it is not central to this paper, we describe it briefly. In general, we allow for multiple inheritance with some restrictions. As to the object types, we demand that the sets of attribute names of the super types are pairwise disjoint. This way we avoid known problems with multiple attributes of the same name. Note that the service type inheritance is completely handled by the Planics parser, so the planners obtain already computed final specification of the service types.

2.2. Comparison with OWL-S and WSMO

It is important to discuss similarities and differences between Planics and two major solutions for the Semantic Web paradigm: OWL-S and WSMO [51]. OWL-S represents semantic information as an OWL ontology. The upper Service ontology links to: Profile explaining "what the service does", Service Model (SM) describing "how the service works", and

Grounding clarifying "how to access it" by mapping SM to the concrete WSDL document. The Profile, similarly to the Planics service type, exposes Inputs, Outputs, Preconditions, and Effects of the OWL-S service. That is, both the approaches use an implicit capability representation, i.e., a service is described by a state transformation. On the other hand, Service Model and Grounding are covered by the process of service registration in the Planics Service Registry.

There are several approaches to service composition built on the top of the OWL-S concept. In [52] Klusch et.al. introduce the OWLS-Xplan toolset converting a composition problem in OWL-S to its equivalent in PDDL (Planning Domain Definition Language) [53] and using the Xplan tool to solve it. The Xplan is a hybrid planner combining the advantages of Hierarchical Task Network (HTN) planners (such as SHOP2 [54]) and action planners (like, e.g., FastForward [55]). The HTN approach relies on decomposition rules allowing to break complex actions into a set of atomic ones. The action based planners are able to find plans consisting of simple actions. Thus, Xplan works best for problems which are in part hierarchically structured.

A hybrid approach to the semantic service matching problem which consists in finding services of input and output coherent with the query is reported in [49]. Comparing to Planics, a single matchmaking step corresponds to finding a service type which can be used to build a plan, regardless on pre- and post-conditions. In other words, matchmaking can be viewed as an OWL-S equivalent for a simplified abstract planning, e.g., planning in types.

As far as an WSMO specification is concerned, it is divided into four parts: Ontologies, Web Services, Goals, and Mediators. Ontologies provide terminology for other WSMO elements. Goals represent the aim of the service composition. The Web Services part defines the web service capabilities in terms of preconditions, postconditions, assumptions, and effects. Mediators are connectors linking the components used to modelling the service. They resolve heterogeneity problems between the components by defining appropriate mappings and transformations.

Since the WSMO concept is much broader than Planics, it is not easy to compare with each other. Concerning the description languages, that of WSMO is far more rich than Planics's, since we are focused on specifying only the service features useful in the planning process. However, WSMO does not provide its own composition method. Since WSMO exploits various relations between concepts, expressed in the RDF-manner [56], the planning problem can be solved by reasoning in the ontology. There exist many tools (reasoners) that can be used for this purpose. Unfortunately, the growth of the number of services and sorts of data exchanged makes the model exponentially hard to analyse. In contrast, the solutions of our approach are "composition oriented", i.e., the tool implements the original composition method. The problem of operating in environments containing a large number of services is handled by a graphDB-based reduction (see Sec. 6) and by dividing composition into the abstract and concrete planning phases.

However, neither OWL-S nor WSMO have gained wide popularity, because they are too complex and not easy to understand [57]. For most enterprises a simpler approach, like, e.g., Semantic Annotations for WSDL (SAWSDL) [58], is often sufficient. Notice that Planics is closer to OWL-S and WSMO than to SAWSDL, as it offers more than a way of annotating WSDL messages. It is similar to OWL-S and differs from WSMO as it proposes a strict way of defining ontologies rather then a general conceptual framework. Compared to both these approaches, Planics is less expressive but at the same time also significantly less complex, thus addressing the cited above shortcomings. Its logic language for expressing preconditions and results is even simpler than the most restricted language used in OWL-S, i.e., OWL-Lite. On the other hand, in Planics the decidability is guaranteed by the construction of its semantics. Summarizing, Planics has its background in formal verification, what means that it is focused on performance rather than expressibility. WSMO is even more complex than OWL-S, and none of the concepts it introduces as an extension of OWL-S is used in Planics. For example, the latter uses no mediators, heavily applied in WSMO to link heterogeneous components. Some parts of WSMO are not specified, and the promises about providing the full semantics for each part have never materialised. The unique feature of Planics is the two-phase planning algorithm, distinguishing between the abstract and the concrete planning.

2.3. Basic Definitions

APP exploits the service and object types of the ontology. Below, we formalize the main concepts introduced in the previous subsection. Note that the *Thing* class is a root concept required by the OWL language, but useless in the abstract planning process, and thus we omit it when not required.

Object types. Let $\mathbb{I}$ be the set containing all *identifiers* used as the names of the attributes, the types, and the objects. We identify the attributes with their names, because during the abstract planning the types of the attributes are irrelevant. Thus, by $\mathbb{A}$ we denote the set of all attributes, where $\mathbb{A} \subset \mathbb{I}$. Then, we define an *object type* to be a pair $(t, Attr)$, where $t \in \mathbb{I}$, and $Attr \subseteq \mathbb{A}$. Moreover, let $\mathbb{T}$ be the set of object types, i.e., all descendants of the classes *Artifact* and *Stamp*.

We define also a transitive, irreflexive, and antisymmetric *inheritance* relation $Ext \subseteq \mathbb{T} \times \mathbb{T}$, such that $((t_1, A_1), (t_2, A_2)) \in Ext$ iff $t_1 \neq t_2$ and $A_1 \subseteq A_2$. Thus, a subtype contains the attributes of a supertype, but more attributes may be introduced.

Example 1. *Consider the object types depicted in Fig. 1. We have* $(Artifact, Ware) \in Ext$ *(i.e., Ware is a subclass of Artifact), as the set of attributes of Artifact is included in that of Ware. Similarly,* $\{(Ware, Boards),$ $(Ware, Nails), (Ware, Arbour), (Stamp, PriceStamp)\} \subseteq Ext$. *Moreover, from transitivity of Ext also* $\{Artifact\} \times \{Boards, Nails, Arbour\} \subseteq Ext$.

By an *object o* we mean a pair $o = (id, type)$, such that $id \in \mathbb{I}$ and $type \in \mathbb{T}$, i.e., a pair containing the object name type. We refer to the elements of this pair respectively by $id(o)$, and $\mathcal{T}p(o)$, for a given object o. Moreover, we denote the set of all objects by $\mathbb{O}$, and define the function $attr : \mathbb{O} \longmapsto 2^{\mathbb{A}}$ assigning a set of the attributes to each object of $\mathbb{O}$.

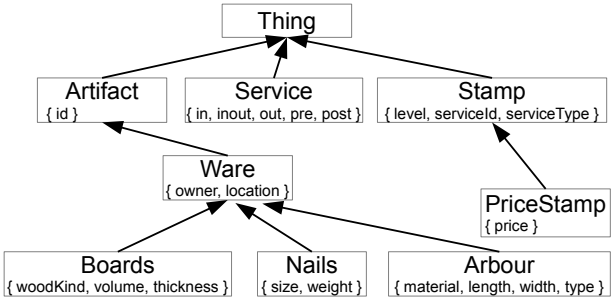


Figure 1: Object types inheritance in an example ontology and introducing new attributes by subtypes.

Service type and user query. The aim of a WSCP process is to compose services to satisfy the user's goal which is formulated as a *user query specification*. The ontology contains *service type specifications* being definitions of the service types that can be used for composition. For abstract planning all these specifications need to be reduced to *abstract forms*, i.e., sets of objects and *abstract formulas* over them.

Definition 1 (Abstract formulas)**.** *The following BNF (Backus-Naur Form) grammar defines an abstract formula over a set of objects O:*

```
<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 grammar above defines DNF (Disjunctive Normal Form) formulas without negations, i.e., alternatives of clauses. Their elements, called *abstract clauses*, are conjunctions of literals. Each non-trivial literal specifies an abstract value of an object attribute, using the appropriate function ($isSet$ or $isNull$). We assume that the abstract formulas used in abstract planning contain no clauses in which $isSet(o.a)$ and $isNull(o.a)$ are applied to the same $o \in O$ and $a \in attr(o)$.

Below, we give the specification syntax of the service types and of the user queries.

Definition 2 (Specification)**.** *A specification is a 5-tuple $(in, inout, out, pre, post)$, such that 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, we denote a user query specification q or a service type specification s by $spec_x = (in_x, inout_x, out_x, pre_x, post_x)$, where $x \in \{q, s\}$, resp. We say that the service s *processes* the objects of the lists in_x, $inout_x$, and out_x. The objects of in_x are read-only, these from $inout_x$ may be modified, while out_x stores only new objects (i.e., these which are produces by the service).

At this point, we would like to explain why the post-conditions can be built also of attributes of the *in*-objects, despite they are declared as read-only. Actually, the reason for this is technical and is associated with the procedure reducing the original conditions into abstract ones. This situation is due to constraints comparing attribute values of objects of *in* and $inout \cup out$, for example $o.a >= x.b$, where $o \in in$, $x \in inout \cup out$, $a \in attr(o)$, and $b \in attr(x)$. Thus, we should have an access to the attributes of the *in* objects also from the post-conditions. However, note that after reduction of conditions to the abstract form we do not use or compare the (concrete) attribute values but only the abstract ones using the functions $isSet$ and $isNull$. Although it would be possible to exclude the *in* objects from the abstract post-conditions domain, we have decided to leave them in order to obtain a better consistency between the full and reduced conditions.

An example of simple Selling service can be found in part I of the material online [19], **where we provide also more examples of the main notions introduced in this section.**

The *service types* and *user queries* are interpretations of their specifications. Before introducing their formal definitions we introduce the notions of *worlds* and *valuation functions*.

Definition 3 (Valuations of object attributes)**.** *Consider an abstract formula over $\mathbb{O}$, such that $\varphi = \bigvee_{i=1..n} \alpha_i$, 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:*

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

Moreover, we restrict a valuation function v_{α_i} to a set of objects $O \subset \mathbb{O}$ by defining $v_{\alpha_i}(O) = v_{\alpha_i}\big|_{\bigcup_{o \in O} \{o\} \times attr(o)}$, and use the notation $v_{\alpha_i}(o)$ instead of $v_{\alpha_i}(\{o\})$ when the valuation function v_{α_i} is restricted to a single object and its attributes.

If an abstract formula omits values of some attributes, then the undefined values introduced in Def. 3 occur. This case is quite common for the service composition domain, as the information we deal with can often be incomplete, uncertain, or irrelevant. Besides overcoming this problem, undefined values are also used to represent families of total valuation functions (see below).

Definition 4 (Consistent functions)**.** *Consider sets A, A', B 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., $\forall_{a \in A'}\ f'(a) = f(a)$.*

By $total(f)$ we denote the family of the total valuation functions which are consistent with a partial valuation function f.

Let $\mathcal{V}_\varphi = \bigcup_{i=1}^{n} total(v_{\alpha_i})$ be a *family of the valuation functions* over φ that are consistent over every abstract clause α_i. $\mathcal{V}_\varphi(O) = \bigcup_{i=1}^{n} total(v_{\alpha_i}(O))$ is the restriction of $\mathcal{V}_\varphi$ to the objects of O and their attributes.

From some point of view covering of the undefined values by families of functions brings additional complexity. However, the aim of introducing this mechanism is twofold. Firstly,

taking into account incompleteness and vagueness of the user query. We believe that the user's knowledge can be limited as she is often unable to precisely describe the state of all objects. Besides, the user probably is focused on the most important features only, leaving the other attributes unspecified. Secondly, in the abstract planning phase we aim at maximizing a possibility of finding a plan, even if it turns out to be impossible to realize it in the next phases, after the real values of the attributes are taken into account. Thus, we consider all combinations of the possible abstract values of the attributes which are left unspecified. Note that for symbolic methods this is not very expensive.

The next definition we introduce is the one of *worlds*:

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., $|w| = |O_w|$.*

The intuition is that a world stands for a state of a set of objects. We introduce also the notion of a *sub-world* of a world w to be a world built from a subset of O_w and v 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 *worldset*. More formally, 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 $(O, \mathcal{V}(O))$ means the set $\{(O, v_i(O)) \mid 1 \leq i \leq n\}$, for $n \in \mathbb{N}$. Intuitively, we distinguish between worldsets and *sets of worlds*, because a worldset is built over the same set of objects but with different valuations (belonging to the same family of functions), while a set of worlds may contain arbitrary worlds. Finally, by $\mathbb{W}$ we denote the set of all worlds. It should be noticed that a valuation function used to define a world is always restricted to the set of the objects of the world, even if this is not stated explicitly.

Now, we define a *user query* and a *service type* as an interpretation of a corresponding specification.

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

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

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

Additionally, we assume that for every user query specification q we have $inout_q \cup out_q \neq \emptyset$ (otherwise planning would become trivial). We use the symbol $\mathbb{S}$ for the set of all service types of the ontology (notice that the sets $\mathbb{S}$ and $\mathbb{T}$ are disjoint). For a service type (W^s_{pre}, W^s_{post}), W^s_{pre} is called the *input worldset*, while W^s_{post} - the *output worldset*. In turn, for a user query (W^q_{pre}, W^q_{post}), W^q_{pre} is called the *initial worldset*, denoted additionally by W^q_{init}, while W^q_{post} is called the *expected worldset* and denoted additionally by W^q_{exp}. It should be also noticed that out_x contains only new objects (i.e., the ones which are absent in W^x_{pre}, but present in W^x_{post}). In the 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. 2.5).

2.4. Abstract Planning Overview

APP aims at composing service types to satisfy a user query. The query specifies two worldsets: an initial one, consisting of the objects owned by the user, and an expected one, containing objects required to be produced as a result of the service composition. In order to define how this is achieved, we need to introduce several auxiliary concepts. The first such a notion is that of compatibility of object states and worlds.

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., $\mathcal{T}p(o) = \mathcal{T}p(o')$ or $(\mathcal{T}p(o), \mathcal{T}p(o')) \in Ext$, and*
- *for all attributes of o, we have that v' agrees with v, i.e., $\forall_{a \in attr(o)} v'(o', a) = v(o, a)$.*

That is, an object of some subtype (o') is compatible with the one of a base type (o), when the valuations of all common attributes are the same.

Example 2. *Consider three objects $o = (w, Ware)$, $o' = (w', Ware)$, and $o'' = (b, Boards)$, valuation functions $v(o), v'(o')$, and partial valuation function $v''(o'')$. Assume that:*

- *$v(o, location) = v'(o', location) = v''(o'', location)$,*
- *$v(o, id) = v'(o', id) = v''(o'', id)$,*
- *$v(o, owner) = v'(o', owner) = v''(o'', owner)$,*

then we have:
$v(o) \succ^{obj} v'(o')$, $v'(o') \succ^{obj} v(o)$, $v''_t(o'') \succ^{obj} v(o)$, and $v''_t(o'') \succ^{obj} v'(o')$, for $v''_t \in total(v'')$.

In order to identify similar worlds we introduce the notion 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 $w' \succ^{wrl} w$, iff there exists a one-to-one mapping $map : O \longmapsto O'$ such that $\forall_{o \in O} v'(map(o)) \succ^{obj} v(o)$.*

The intuition behind this definition is that a world w' is compatible with a world w when their sizes are the same and each object of w has a compatible counterpart in w'. Our planning process requires also identifying similar worlds of different sizes. To this aim we define the notion of *worlds sub-compatibility*.

Definition 9 (Worlds sub-compatibility)**.** *Let w, w' be worlds such that $w = (O, v)$ and $w' = (O', v')$. 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.*

2.5. World Transformations

World transformation is an important notion in our approach. A service type s of specification $spec_s$ can transform a world w, called *world before*, if w is sub-compatible with some input world of s. The resulting world w', called a *world after*, contains the objects of out_s which together with the objects of $inout_s$, are in the states consistent with some output world of s. The states of the other objects of w are not changed. In general there can be more than one world which results from transforming a given world by a service type as more than one sub-world of w can be compatible with an input world of s. This is the reason for introducing a *context function* which, for each service type s, maps the objects of the worlds before and after, and the objects of the input and output worlds.

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*.*

It should be noticed that the above function can be defined if the number of objects in the set O is at least the same as in the union of sets defined by the service type s, i.e., if $|O| \geq |in_s \cup inout_s \cup out_s|$. World transformation is defined below.

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

Let's explain shortly the conditions above. 1. (2., respectively) specify that the *world before* contains a sub-world built over IN (IO, resp.) which is compatible with a sub-world of some input world of s, built over the objects of in_s ($inout_s$, resp.). The objects state of IN (IO, resp.) is consistent with pre_s. 3. (4.) say that the *world after* contains a sub-world built over IO (OU, resp.) which is compatible with a sub-world of some output world of s, built over the objects of $inout_s$ (out_s, resp.). The objects state of IO (OU, resp.) is consistent with $post_s$. 5. expresses that the objects of IO can change their states, only.

Next, we define a sequence of world transformations.

Definition 12 (Transformation sequences)**.** *Let* $seq = ((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k}))$ *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* $\forall_{1 \leq i \leq k}\ w_{i-1} \overset{s_i, ctx^{s_i}_{O_i}}{\rightarrow} w_i = (O_i, v_i)$ *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* $\mathbb{S}^\star$*.*

Quasi-transformation sequences. Let $seq = ((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k}))$ be a sequence of length k, where, for $1 \leq i \leq k$ we have $s_i \in \mathbb{S}$, $O_i \subseteq \mathbb{O}$, and $ctx^{s_i}_{O_i}$ is a context function. Then, *seq* is a *quasi-transformation sequence for w*, if there is $1 \leq j < k$ such that $((s_1, ctx^{s_1}_{O_1}), \ldots, (s_j, ctx^{s_j}_{O_j}))$ is a transformation sequence for w. By the *q-length* of *seq* we mean such a maximal j, whereas the *executable prefix of* seq (denoted seq_E) is the prefix of *seq* of length j. The *final world* of *seq* of q-length is the world obtained by transformation of w by seq_E. So, *seq* is a quasi-transformation sequence for w if it is not a transformation sequence, but some its non-empty prefix is so.

After defining the transformation sequences we are ready to give definitions of *user query solutions* or simply *solutions*, as a step towards defining 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 (or a q-solution), 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, a solution of q is a transformation sequence which transforms some initial world of q to a world sub-compatible to some expected world of q.

2.6. Plans

Using the definition of a solution to the user query q we can introduce the notion of an (abstract) plan. By such a plan we mean an equivalence class of the solutions of q that are built over the same service types. This means that we neglect the ordering of service type occurrences, which has two advantages: 1) the user receives essentially different plans, and 2) plans are exponentially more compact than solutions. In order to define plans we need an equivalence relation on the solutions.

Definition 14 (Equivalence of user query solutions)**.** *Let the function count* $: \mathbb{S}^\star \times \mathbb{S} \longmapsto \mathbb{N}$ *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* $s \in \mathbb{S}$*.*

This means that each two query solutions built over the same number of the service types belong to the same equivalence class. Their contexts are not taken into account.

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 should be noticed that all the solutions within an abstract plan are built over the same *multiset* of service types. By the *length of an abstract plan* $[seq]_{\sim}$ we mean the length of its representative, i.e., $length(seq)$.

Example 3. *Assume, we have an ontology containing the following service types: $Select$, $Selling$, $Transport$, and $WoodBuilding$, while the object types are shown in Fig. 5. Let $Select$ be a service type corresponding to selecting wares to be bought, and let $Selling$ change the owner of a given ware. The $Transport$ service type is able to change the location of some ware, while $WoodBuilding$ produces an arbour using all the available boards and nails.*

Assuming that the user wants to have an arbour and an appropriate query is specified, the shortest abstract plan is represented by the multiset $M_1 = [Select, Selling]$. This plan consists of one solution only 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 using the $WoodBuilding$ service type is represented by the multiset $M_3 = [Select : 2, Selling : 2, WoodBuilding]$, 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, these are the sequences $(Select, Select, Selling, Selling, WoodBuilding)$ and $(Select, Selling, Select, Selling, WoodBuilding)$.

An extended version of this example, including the full formalisation, can be found in part I of [19] (Example 13).

It should be noticed that the set of transformation sequences built over a multiset of service types can include not only the sequences having different orders, but also having different contexts. More precisely, if a world contains several similar objects even the same linearisation of service types can lead to a slightly different result when it is applied to different object instances. To see an example, consider an extended solution of Example 3, i.e., $(Select, Select, Selling, Selling, Transport, WoodBuilding)$. There are two transformation sequences which match the solution, i.e., the ones in which either boards or nails are transported. This difference does not need (but can) be important at the stage of abstract planning, but definitively it matters in the next planning phases, when the concrete values are taken into account.

In order to cope with this problem we developed an algorithm which uses the combinatorial structure of a given multiset and, abstracting from the object attributes, browses the space of all potential solutions taking into account only the indistinguishable ones. Finally, the reported results are validated by checking the attribute valuation and the presumed constraints [59]. The algorithm has been implemented as a module of Planics called MultisetExplorer.

According to the theorem below, APP is a hard problem.

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

Proof. See part II of [19]. □

Notice that if we limit the length of plans to some constant n, then APP is clearly in NP, so it is NP-complete. This follows from the fact that we can non-deterministically generate all the sequences of service types of length n and for each of them check in polynomial time whether it is a solution.

3. SMT-based Approach

In this section we present a translation of APP to an SMT formula, which is checked for satisfiability by an SMT-solver. 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 present an overview of our planning algorithm, and show how to construct the formula φ_k^q corresponding to APP. Then, we discuss the symbolic object and world representations, followed by the encoding of the components of φ_k^q.

3.1. Introduction to SMT-LIB v2 language

The main motivation for defining the SMT-LIB language is 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 [60] by Ranise and Tinelli, but its succesive revisions led to SMT-LIB version 2, which was announced in 2010 [61]. 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 APP introduced in the previous section, we build a formula over Boolean and integer variables. Then, this formula is given to a solver to check whether the formula is satisfiable, i.e., whether there is an interpretation of the variables used that evaluates the formula to **true**.

An example introducing several basic SMT-LIB v2 commands is given in part III of [19]. A comprehensive tutorial on SMT-LIB v2 can be found in [62].

3.2. Abstract Planning Algorithm

The core of our SMT-based abstract planner is an adaptation of a symbolic Bounded Model Checking (BMC) method [63]. The main idea behind BMC is to search for a finite execution of a system satisfying a required property. In the case of APP, given an ontology, a user query q, and some additional parameters k_{min} and k_{max}, the planner seeks for solutions of length k, where $k_{min} \leq k \leq k_{max}$. The algorithm starts with $k = k_{min}$ and encodes APP as an appropriate SMT formula φ_k^q which is checked for satisfiability. If the solver returns *SAT*, then a solution, i.e., a representative of some abstract plan, has been found. In order to eliminate the other solutions representing the same abstract plan, a *blocking formula* is computed (see Section 3.6). When the tested formula is unsatisfiable, it means that there are no other plans of length k. Then, until k does not exceed k_{max}, k is incremented, and the planner looks for a possibly longer

plan. Overall, the following SMT-formula φ_k^q encodes the plans of length k satisfying the query q:

$$\varphi_k^q = \mathcal{I}^q \wedge \bigwedge_{i=1..k} \Big(C_i \wedge \bigvee_{s\in\mathbb{S}} \mathcal{T}_i^s\Big) \wedge \mathcal{E}_k^q \tag{1}$$

where $\mathcal{I}^q$ and $\mathcal{E}_k^q$ are formulas encoding the initial and the expected worldsets, respectively, C_i encodes the i-th context function, and $\mathcal{T}_i^s$ encodes worlds transformation by a service type s.

In order to find all plans of length k we check the satisfiability of the conjunction $\varphi_k^q \wedge \mathcal{B}_k^q$, where $\mathcal{B}_k^q$ stands for a blocking formula (see Section 3.6).

3.3. Objects and Worlds

The formulas mentioned in the previous subsection are built over *variables* which have to be first allocated in the SMT-solver memory. These variables are organized in structures representing objects and worlds, called *symbolic objects* and *symbolic worlds*, respectively. A symbolic object is represented by an integer variable encoding its type, called a *type variable*, and a number of Boolean variables to represent the object attributes, called the *attribute variables*. In order to encode the identifiers and types as numbers, we introduce an auxiliary function $num : \mathbb{A} \cup \mathbb{T} \cup \mathbb{S} \cup \mathbb{O} \longmapsto \mathbb{N}$, which assigns a natural number to every attribute, object type, service type, and object.

Each symbolic world, consisting of a number of symbolic objects, is indexed by a natural number from 0 to k. The i-th symbolic object belonging to the j-th symbolic world is a tuple: $\mathbf{o}_{i,j} = (\mathbf{t}_{i,j}, \mathbf{a}_{i,0,j}, \mathbf{a}_{i,1,j}, ..., \mathbf{a}_{i,max_{at}-1,j})$, such that $\mathbf{t}_{i,j}$ is a type variable, $\mathbf{a}_{i,x,j}$ is an attribute variable for $0 \le x < max_{at}$, where max_{at} is the maximal number of the attribute variables needed to represent the object. It is important that each symbolic world represents *a set of worlds*, and we obtain a single world when we interpret its variables according to some *valuation*. 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. 2 shows subsequent symbolic worlds of a transformation sequence.

3.4. User Query

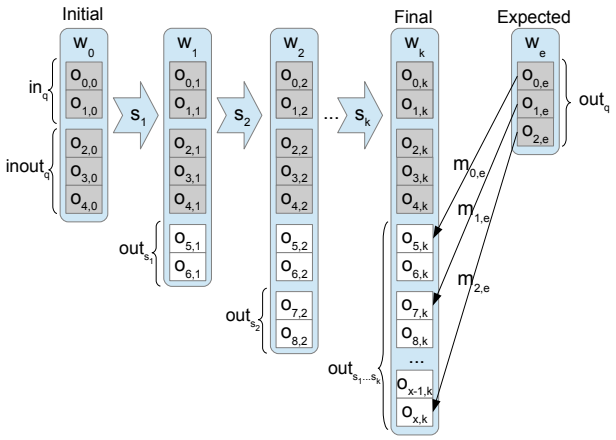


Figure 2: Symbolic worlds

In order to encode the worldset W_{init}^q by a symbolic world $\mathbf{w}_0$, the variables used to represent the objects of $in_q \cup inout_q$ are declared. Then, using these variables, the formula $\mathcal{I}^q$ is built. It encodes the types and the states of the objects of the initial worldset:

$$\mathcal{I}^q = tpF(\mathbf{w}_0, in_q \cup inout_q) \wedge stF(\mathbf{w}_0, W_{init}^q) \tag{2}$$

The formula $tpF(\mathbf{w}_i, O)$ encodes the types of the objects of the set O over a symbolic world $\mathbf{w}_i$:

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

Next, we define the formula $stF(\mathbf{w}_i, W)$ which is used to encode the states of the objects of the worldset $W = (O, \mathcal{V})$ 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 the expression $vF(\mathbf{w}_i, v, o, a)$ encodes the valuation v of the attribute $o.a$ over the variables constituting the symbolic world $\mathbf{w}_i$. It is defined as follows:

$$vF(\mathbf{w}_i, v, o, a) = \begin{cases} \mathbf{a}_{num(o),num(a),i}, & \text{if } v(o,a) = \textbf{true}, \\ \neg\mathbf{a}_{num(o),num(a),i}, & \text{if } v(o,a) = \textbf{false}, \\ \textbf{true}, & \text{if } v(o,a) \text{ is undef.} \end{cases} \tag{3}$$

Thus, the symbolic world $\mathbf{w}_0$ represents the initial worldset. Then, after the first transformation we obtain the symbolic world $\mathbf{w}_1$, enriched by the objects produced during the transformation (see Fig. 2). 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 *final worlds* that are possible to obtain after k transformations of the initial worldset. 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 $W_{exp}^q = (in_q \cup inout_q \cup out_q, \mathcal{V}_{post}^q)$. 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 worldset, we know their indices, and therefore their states are encoded by the formula $ioExp$, defined as follows:

$$ioExp(\mathbf{w}_k, W_{exp}^q) = \\ stF\Big(\mathbf{w}_k, (in_q \cup inout_q, \mathcal{V}_{post}^q(in_q \cup inout_q))\Big), \tag{4}$$

where $\mathcal{V}_{post}^q(in_q \cup inout_q)$ is the family of the valuation functions $\mathcal{V}_{post}^q$ 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 worldset 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 $\mathbf{w}_k$, 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 (see Fig. 2). We encode their states by the formula $outExp$:

$$outExp(\mathbf{w}_e, W^q_{exp}) = stF\Big(\mathbf{w}_e, (out_q, \mathcal{V}^q_{post}(out_q))\Big), \tag{5}$$

where $\mathcal{V}^q_{post}(out_q)$ 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 $\mathbf{w}_i$:

$$sbF(\mathbf{w}_i, O) = \bigwedge_{o \in O} \bigvee_{t \in subT(o)} \mathbf{t}_{num(o),i} = t \tag{6}$$

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 worldset, we need a mapping between the objects from $\mathbf{w}_e$ and those from a final world $\mathbf{w}_k$, produced by the consecutive transformations. Thus, in the symbolic world $\mathbf{w}_e$, we allocate $p = |out_q|$ additional *mapping variables*, denoted by $\mathbf{m}_{0,e}, \ldots, \mathbf{m}_{p-1,e}$. These variables store the indices of the objects from a final world compatible with the objects encoded over $\mathbf{w}_e$. Thus, the last part of the expected worlds encoding is the formula:

$$mpF(\mathbf{w}_e, \mathbf{w}_k) = distinct(\mathbf{m}_{0,e}, \ldots, \mathbf{m}_{p-1,e}) \wedge \bigwedge_{i=0}^{p-1} \bigvee_{j=|\mathbf{w}_0|}^{|\mathbf{w}_k|-1} (eqF(\mathbf{o}_{i,e}, \mathbf{o}_{j,k}) \wedge \mathbf{m}_{i,e} = j) \tag{7}$$

where $distinct(i_1, \ldots, i_n)$ is a formula encoding pairwise inequality of variables $i_1, \ldots, i_n$, i.e., it is true only when every variable is assigned a different value.

For example, the arrows from the expected symbolic world to the final one in Figure 2 depict an exemplary valuation of the mapping variables, where $\mathbf{m}_{0,e} = 5$, $\mathbf{m}_{1,e} = 7$, and $\mathbf{m}_{2,e} = x$.

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

$$\mathcal{E}^q_k = ioExp(\mathbf{w}_k, W^q_{exp}) \ \wedge \ sbF(\mathbf{w}_e, out_q) \wedge outExp(\mathbf{w}_e, W^q_{exp}) \ \wedge \ mpF(\mathbf{w}_e, \mathbf{w}_k) \tag{8}$$

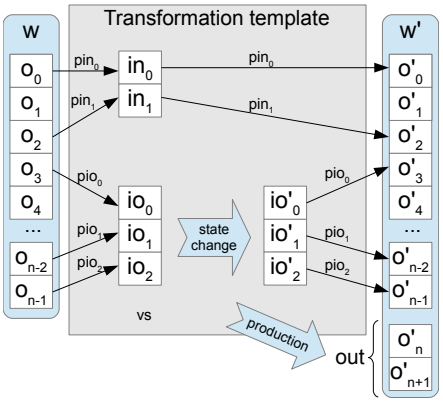


Figure 3: Transformation template

3.5. World Transformation

Following Def. 11, given a world w, a service type s, and a context function, the world after the transformation w', can be computed. In the previous subsection we presented the encoding of the initial and the expected worldsets. Now, our aim is to show how to construct a formula which encodes all possible transformations $w \xrightarrow{s} w'$ over two subsequent symbolic worlds $\mathbf{w}_i$ and $\mathbf{w}_{i+1}$, and then to extend the encoding on the consecutive pairs of intermediate symbolic worlds. If this formula is satisfiable, then from the obtained valuation we can identify the subsequent service types and the context functions.

For each transformation, we introduce a *transformation template* (see Fig. 3) which consists of the three sets of symbolic objects $\mathbf{in}_i$, $\mathbf{io}_i$, and $\mathbf{io}'_i$, the two sets of integer mapping variables $\mathbf{pin}_i$ and $\mathbf{pio}_i$, and the additional integer variable $\mathbf{vs}_i$. The type of a service which transforms the worlds is encoded by $\mathbf{vs}_i$. Its input worlds are represented by symbolic objects of $\mathbf{in}_i$ and $\mathbf{io}_i$, and they strictly correspond to the sets IN and IO of Def. 11. The variables of $\mathbf{pin}_i$ and $\mathbf{pio}_i$ are used to encode the context functions, i.e., mappings between symbolic objects from $\mathbf{in}_i$, $\mathbf{io}_i$, $\mathbf{io}'_i$ and symbolic objects from w and w' (see Fig. 3).

Finally, the modified objects are encoded using variables of $\mathbf{io}'_i$, and the new objects, i.e., these produced during the transformation, are encoded directly over the resulting symbolic world, because we know a-priori their indexes. Moreover, for every object $o_{i,j} \in \mathbf{w}_i$ we introduce a pair of the auxiliary Boolean variables $(\mathbf{isIn}_{i,j}, \mathbf{isIo}_{i,j})$. At most one of them evaluates to *true* if the j-th object is the value of the i-th context function: $\mathbf{isIn}_{i,j}$ is *true* if the j-th object is taken as read-only (*in*) object, or $\mathbf{isIo}_{i,j}$ is *true* if it plays a role of one of the *inout* objects.

We take advantage of the fact that several aspects of the worlds transformations can be encoded in the same way, regardless of the service type. Thus, the formula C_i responsible for assigning objects from worlds to the i-th transformation template is as follows:

$$C_i = distinct\Big(\{\mathbf{p} \mid \mathbf{p} \in \mathbf{pin} \cup \mathbf{pio}\}\Big) \wedge \bigwedge_{o_{i,j} \in \mathbf{w}_i} (\neg\mathbf{isIn}_{i,j} \vee \neg\mathbf{isIo}_{i,j}) \wedge Ain(o_{i,j}) \wedge Aio(o_{i,j}) \tag{9}$$

where:

- the *distinct* clause ensures that every object from **w** is assigned to at most one object from the transformation template,
- the disjunction of the negated variables **isIn** and **isIo**, which should be satisfied for each object $o_{i,j}$, states that the object is not assigned as *in* object, or is not assigned as *inout* object of the transformation template,
- the $Ain(o_{i,j})$ and $Aio(o_{i,j})$ formulae encode the assignments of the objects from the worlds to the transformation template. They are defined as follows:

$$Ain(o_{i,j}) = (\neg \mathbf{isIn}_{i,j} \wedge \bigwedge_{k=0}^{max_{in}-1} \mathbf{pin}_{i,k} \neq j) \vee \Big(\mathbf{isIn}_{i,j} \wedge \bigvee_{k=0}^{max_{in}-1} (eq(o_{i,j}, \mathbf{in}_{i,k}) \wedge \mathbf{pin}_{i,k} = j)\Big) \quad (10)$$

$$Aio(o_{i,j}) = \Big(\neg \mathbf{isIo}_{i,j} \ \wedge \ eq(o_{i,j}, o_{i+1,j}) \ \wedge \bigwedge_{k=0}^{m} \mathbf{pio}_{i,k} \neq j\Big) \vee \Big(\mathbf{isIo}_{i,j} \wedge \bigvee_{k=0}^{m} (eq(o_{i,j}, \mathbf{io}_{i,k}) \wedge \mathbf{pio}_{i,k} = j \wedge eq(\mathbf{io}'_{i,k}, o_{i+1,j}))\Big), \quad (11)$$

for $m = max_{inout} - 1$.

Note that according to Formula (11) every object which is not assigned to *inout* is passed unchanged to the next world. Thus, the formula C_i deals with the i-th context function, as well as with copying the unchanged objects to the next world of the symbolic path.

Now, we encode the part of the i-th world transformation specific to a service of type s:

$$\mathcal{T}_i^s = sRestr(s, i) \wedge sTrans(s, i) \quad (12)$$

where

$$sRestr(s, i) = \bigwedge_{j=0..|in_s|-1} (\mathbf{pin}_{i,j} \geq 0 \wedge \mathbf{pin}_{i,j} < |\mathbf{w}_i|) \wedge \bigwedge_{j=0..|inout_s|-1} (\mathbf{pio}_{i,j} \geq |in_q| \wedge \mathbf{pio}_{i,j} < |\mathbf{w}_i|) \wedge \mathbf{vs}_i = num(s) \quad (13)$$

encodes the restrictions on the number of objects needed by the service type s, and sets the value of the variable $\mathbf{vs}_i$ to the value corresponding to the service type s, and

$$sTrans(s, i) = inF(\mathbf{in}_i, \mathbf{io}_i, W^s_{pre}) \ \wedge ouF(\mathbf{w}_{i+1}, \mathbf{io}'_i, W^s_{post}) \quad (14)$$

encodes the preconditions and the changes introduced by the transformation, according to the specification of service type s. Moreover, *inF* encodes input worldset:

$$inF(\mathbf{in}_i, \mathbf{io}_i, W^s_{pre}) = stF\Big(\mathbf{in}_i, (in_s, \mathcal{V}^s_{pre}(in_s))\Big) \wedge stF\Big(\mathbf{io}_i, (inout_s, \mathcal{V}^s_{pre}(inout_s))\Big) \wedge sbF(\mathbf{in}_i, in_s) \wedge sbF(\mathbf{io}_i, inout_s),$$

and *ouF* stands for output worldset:

$$ouF(\mathbf{w}_{i+1}, \mathbf{io}'_i, W^s_{post}) = sbF(\mathbf{w}_{i+1}, out_s) \wedge stF\Big(\mathbf{io}'_i, (inout_s, \mathcal{V}^s_{post}(inout_s))\Big) \wedge stF\Big(\mathbf{w}_{i+1}, (out_s, \mathcal{V}^s_{post}(out_s))\Big),$$

Theorem 2 (Correctness of the encoding)**.** *The formula φ^q_k is satisfiable iff there is a solution to user query q of length k.*

Proof. See part II of [19]. □

3.6. Multiset Blocking

A blocking formula is the final element of our encoding. It is aimed at eliminating the solutions of the plans already known from a further search. To this aim, a convenient representation of an abstract plan is a multiset of the service types occurring in a user query solution.

We encode counting of the service types instances in transformation sequences, in order to represent multisets. Let the set of all possible sequences of Boolean values be denoted by $\mathbb{B}^\star$, and let $cnt : \mathbb{B}^\star \longmapsto \mathbb{N}$ be a function assigning the number of occurrences of the value **true** to every Boolean sequence. The encoding of this function is 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 *ite* means *if-then-else*, and the expression $ite(b_i, 1, 0)$ returns 1 if b_i equals **true** and 0 otherwise. When a user query solution is found, the sequence of service types $s = (s_1, \ldots, s_k)$ is extracted and its multiset representation $M_s = ((s_1, c_1), \ldots, (s_n, c_n))$ is computed, where $s_i \in \mathbb{S}$, c_i stands for the number of instances of s_i in the sequence, and $1 \leq i \leq n \leq k$. The following formula blocks all solutions built over M_s:

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

Assume that at some point of a computation j abstract plans have been found. The formula $\mathcal{B}^q_k$ which blocks the solutions of j plans (represented by multisets) is as follows:

$$\mathcal{B}^q_k = \bigwedge_{i=1}^{j} block(M_{s_i}) \quad (15)$$

3.7. Experimental Evaluation of Multiset Blocking

Our experimental results are given in the last section and compared with these of other approaches. Here we only evaluate the efficiency of the encoding of the multiset blocking by comparing it with the sequence blocking. To this aim several experiments have been performed, where (15) has been replaced by the formula blocking the user query solutions found so far, regardless of their contexts:

$$\mathcal{B}^q_k = \bigwedge_{l=1}^{j} \Big(\neg \bigwedge_{i=1}^{k} (\mathbf{vs}_i = s_{l,i})\Big) \quad (16)$$

The overall conclusion is that the multiset blocking outperforms the sequence blocking when there is a large number of solutions, e.g., when a single plan has many possible linearisations. Despite the multiset encoding is more expensive than the sequence blocking, it is already more efficient when the number of plan linearisations exceeds several hundreds. However, we compare here only the CPU time consumed by the SMT solver. If we take additionally into account the computation time needed to process every plan found (getting model from the solver, decoding and storing the plan, updating the blocking formula), then the multiset blocking is even more superior than the sequence blocking. Some conclusions of the experimental evaluation of the multiset blocking and a description of the implementation is given in Part III of [19].

4. GA-based Approach

In this section we aim at using Genetic Algorithms for finding abstract plans for a given user query q. We start with presenting the general idea of the approach.

4.1. General Idea

The overall Genetic Algorithm scheme is presented in Alg. 1. An abstract plan to be found is modelled by an individual, each gene of which corresponds to a service type. The number of the genes of an individual and the number of service types in the abstract plan are therefore equal.

In each iteration of GA, individuals are selected using the roulette selection operator for a temporary population. Next, they are mixed using standard one point crossover and then mutated taking advantage of our mutation operator (see Sec. 4.5). A temporary population becomes a current population in the next iteration of GA. At the end of the iteration all individuals are evaluated using a fitness function (see Sec. 4.4).

Genetic Algorithm(N, I, Cp, Mp)
Input: number of individuals in population: N, number of iterations: I, crossover probability: Cp, mutation probability: Mp
Result: an individual with the highest fitness value

```
begin
    P ← generateInitPop(N) ; // the initial population
    evaluate(P) ;            // evaluate individuals of P
    for i ← 1..I do
        T ← selection(P) ;     // a temporary population
        T ← crossover(T);    // apply genetic operators
        T ← mutation(T);     // w.r.t. Cp and Mp values
        P ← T ;              // a new population obtained
        evaluate(P);
    bi ← findBI(P) ;         // find the best individual
    return bi
```

Algorithm 1: The general scheme of GA

The fitness function for an individual is calculated for each possible combination of an initial and an expected world. Then, the maximum value of all these calculations is set for an individual. This allows to obtain abstract plans related to any pair of an initial and an expected world.

While GA maintains a population of individuals, each represented by a multiset M of service types (rather than a sequence of them), it is necessary to test whether M really corresponds to an abstract plan. This is done by running the procedure *seqGen* (see Sec. 4.3) for M and an initial state w. If the resulting sequence $seq_{(M,w)}$ is a solution to the user query q, then M represents an abstract plan. In what follows we present our approach in detail.

4.2. Encoding Abstract Plans

The first step towards the encoding consists in assigning a unique natural number to each service type defined in the ontology and defining an individual to be a multiset over the set of the above numbers. The initial population for GA is a set of randomly generated individuals, which implies that at the beginning of the algorithm the population contains multisets of service types which do not necessarily represent abstract plans. The non-standard form of a GA individual, in which we do not care about the order of the genes, allows for performing genetic operations in such a way that we do not have to receive offspring containing service types in the correct order. However, before applying the fitness function the multiset is transformed into a sequence of service types. The fact that we do not generate all the sequences results in a substantial reduction of the state space, which allows to obtain user query solutions even in search spaces of sizes greater than 2^{100} (see Sec. 7).

4.3. From a Multiset to a Sequence

In order to compute the fitness function value for an individual we need to generate a transformation sequence built over the elements of its multiset. The sequence should be a user query solution, and therefore we are interested only in these sequences which transform some initial world. Due to the fact that the individuals should be evaluated against all the combinations of the initial and expected worlds, the procedure *seqGen* (given below), aimed at obtaining such sequences from a multiset, is applied later to all the initial worlds.

Alg. 2 iterates, building a resulting sequence in the following way: it chooses a service type s belonging to the multiset M and able to transform[1] a current world w_f, and removes s from M appending it to the resulting sequence instead. The process is started for w_f being an initial world $w \in W^q_{init}$; in further iterations the current world becomes the one obtained by transforming w_f by s chosen recently. If M contains no more service types which can be executed in the current world, then the elements which remained in M are appended in a random order at the end of the sequence. Besides the sequence $seq_{(M,w)}$ the procedure returns also a world w_f and a natural number l, where w_f is the final world of $seq_{(M,w)}$, and l is the (q-)length of $seq_{(M,w)}$ if $seq_{(M,w)}$ is a (quasi-)transformation sequence, respectively. The

[1] If there are more than one such a service type, then one of them is chosen randomly.

Procedure seqGen(M, w)
Input: a multiset of service types: M, an initial world: w
Result: a triple $(\mathbf{seq_{(M,w)}}, \mathbf{l}, \mathbf{w_f})$, where $seq_{(M,w)}$ is a (quasi) transformation sequence, l is the (q-)length of $seq_{(M,w)}$, and w_f stands for the final world of $seq_{(M,w)}$

begin
  $seq_{(M,w)} \leftarrow \epsilon$; // empty sequence
  $l \leftarrow 0$;
  $w_f \leftarrow w$;
  while M **contains** s **that can be executed in** w_f **do**
    $seq_{(M,w)} \leftarrow seq_{(M,w)} \cdot s$; // append s to $seq_{(M,w)}$
    $l \leftarrow l + 1$;
    $M \leftarrow M \setminus \{s\}$; // remove s from M
    $w_f \leftarrow transform(w_f, s)$;
  while M **is not empty do**
    $M \leftarrow M \setminus \{s\}$; // remove some s from M
    $seq_{(M,w)} \leftarrow seq_{(M,w)} \cdot s$; // append s to $seq_{(M,w)}$
  return $(seq_{(M,w)}, l, w_f)$

Algorithm 2: Proc. *seqGen* generating a sequence from a multiset

above results are used later to compute the fitness value of the individual.

It should be also explained why not all sequences which can be constructed from a given multiset M are taken into account. The first reason is a large number of all possible sequences, i.e., $k!$ for M of cardinality k. Moreover, we clearly give priority to the sequences which transform some initial world of the user query. The final reason is that constructing another sequence from the same multiset is still possible if an individual passes to the next generation.

4.4. Fitness Function

In order to evaluate an individual we need to calculate its fitness value. Intuitively, the value of the fitness function for a given individual corresponds to how close the individual is to a solution. This means that the fitness value of an individual representing a solution is the greatest. So, by iteratively generating new sets of individuals having bigger fitness values than their "predecessors", the algorithm is trying to obtain user query solutions. The selection operator of GA selects individuals for genetic operators according to their fitness values. This means that the bigger fitness value of an individual the more likely it is selected for the next iteration of GA.

Before we get to the details, we introduce first the notion of a *good service type*. Intuitively, a service type is good if it produces objects of types that either can be a part of some expected world, or can be an input for another good service type. To define this formally, assume that q is a user query, and M is a multiset of k service types such that $seq_M = (s_1, \ldots, s_k)$ is a sequence of the service types of M. Consider two different service types s_i and s_j, where $i, j \in \{1, \ldots, k\}$, and denote by in_{s_i}, $inout_{s_i}$, and out_{s_i} respectively the sets of the objects used, modified, and produced by the service type s_i, and by $inout_q$ and out_q - the objects requested to be modified and produced by the user query q. Next, we define the function $\mathcal{T}p^{\downarrow} : 2^{\mathbb{O}} \longmapsto 2^{\mathbb{T}}$ which $\mathcal{T}p^{\downarrow}$ assigns a set of the types as well as their subtypes to a set of objects O. Formally:

- $\mathcal{T}p^{\downarrow}(O) = \{t \in \mathbb{T} \mid \exists o \in O : t = \mathcal{T}p(o) \vee (\mathcal{T}p(o), t) \in Ext\}$,

We say that s_i is a *good service type* for the sequence seq and the user query q, if $\mathcal{T}p^{\downarrow}(inout_{s_i} \cup out_{s_i}) \cap \mathcal{T}p^{\downarrow}(inout_q \cup out_q) \neq \emptyset$, or there exists s_j in seq, such that s_j is a good service type and $\mathcal{T}p^{\downarrow}(inout_{s_i} \cup out_{s_i}) \cap \mathcal{T}p^{\downarrow}(in_{s_j} \cup inout_{s_j}) \neq \emptyset$. A set of the good service types for a user query q and a multiset M is computed by the procedure GST presented as Alg. 3.

Next, for a triple $(seq_{(M,w)}, l, w_f)$ returned by $seqGen(M, w)$ and an expected world w_q, the level of usefulness of an individual is calculated according to Eq. 17:

$$U_{seq_{(M,w)},l,w_f,w_q} = \frac{|w'_{f,q}| * \delta + c_{f,q}^{out_q} * \alpha + l * \beta + g_{seq_{(M,w)}} * \gamma}{|w_q| * \delta + |out_q| * \alpha + k * \beta + k * \gamma} \quad (17)$$

where:

- $|w'_{f,q}|$ is the number of the objects in $w'_{f,q}$ which is the maximal sub-world of w_f compatible with a subworld of w_q,
- $c_{f,q}^{out_q}$ is the number of the objects from w_f whose types are consistent with the types of the objects from w_q and included in the out_q list of the user query q,
- $g_{seq_{(M,w)}}$ is the number of the good service types in $seq_{(M,w)}$,

Procedure GST(M, q)
Input: a multiset of service types: M, a user query: q
Result: a set of good service types occurring in M: GS
begin
  $GS \leftarrow \emptyset$;
  $S \leftarrow \overline{M}$; // set of all types occurring in M
  while $S \neq \emptyset$ **do**
    $s \leftarrow x \in S$; // an arbitrary element of S
    $S \leftarrow S \setminus \{s\}$; // remove s from S
    if $\mathcal{T}p^{\downarrow}(inout_s \cup out_s) \cap \mathcal{T}p^{\downarrow}(inout_q \cup out_q) \neq \emptyset$ **then**
      $GS \leftarrow GS \cup \{s\}$; // add s to GS
  $S'' \leftarrow GS$;
  repeat
    $S' \leftarrow \emptyset$;
    $S \leftarrow \overline{M} \setminus GS$;
    while $S \neq \emptyset$ **do**
      $s \leftarrow x \in S$; // an arbitrary element of S
      $S \leftarrow S \setminus \{s\}$; // remove s from S
      foreach $g \in S''$ **do**
        if $\mathcal{T}p^{\downarrow}(inout_s \cup out_s) \cap \mathcal{T}p^{\downarrow}(in_g \cup inout_g) \neq \emptyset$ **then**
          $S' \leftarrow S' \cup \{s\}$; // add s to S'
          break;
    $GS \leftarrow GS \cup S'$; // add S' to GS
    $S'' \leftarrow S'$;
  until $S' = \emptyset$;
  return GS

Algorithm 3: Proc. GST computing a set of good service types in a multiset M for a given user query q

- $|out_q|$ is the number of the objects in the set out_q list of the user query q,
- k is the length of $seq_{(M,w)}$,
- $\alpha, \beta, \delta, \gamma$ are the parameters of the fitness function, explained below.

In all the experiments presented in Sec. 7 the following values of the parameters are used: $\alpha = 5, \beta = 4, \gamma = 4$, and $\delta = 1$. An inspiration for such values of the parameters is the evolutionary process of searching abstract plans. The values of α, β, and γ are significantly greater than value of δ, in order to better assess the individuals having more good services, longer executable prefix, or providing objects satisfying the user query. The values of all parameters have been evaluated experimentally. We have made dozens of experiments changing slightly one value at a time. In order to tune the parameter values we have used several additional benchmarks not reported in Sec. 7. After the results have been compared, we found these values optimal[2].

After GA finds a user query solution and stores it in memory, its next task is to ensure that other solutions remaining in the search space can be found. Moreover, each individual representing a solution equivalent to one already found should be eliminated. The basis of the elimination is the *measure of similarity* between the plans found so far and the individual evaluated. The measure grows together with the number of service types which are common for the multiset considered and one of the plans stored in the memory.

The measure of similarity between a multiset M and a nonempty set Sol of plans (given by multisets) is computed as

$$sim_M^{Sol} = max\left(\left\{ \frac{|M \cap S|}{|M|} \;\middle|\; S \in Sol \right\}\right) \tag{18}$$

where the operation $\cap$ is applied to multisets.

The resulting measure computed for a multiset identical to some plan of Sol is equal to 1, while the result 0 is returned for a multiset built over a set of service types completely different from these constituting the plans found so far.

Finally, when there are already some solutions found, the fitness value of the individual M for a given initial world w and an expected world w_q is calculated according to Eq. 19:

$$fit_{M,w,w_q} = \frac{U_{seq_{(M,w)},l,w_f,w_q}}{D^{(sim_M^{Sol})}}. \tag{19}$$

GA is supposed to find many different abstract plans in one run. If an individual represents an abstract plan which has been already found, then its fitness is decreased. Different abstract plans may vary by a few service types only. This leads to the following requirement. While an evolutionary process finds subsequent good services, building a new solution is essential, but at the same time the algorithm should prevent from obtaining a currently known abstract plan. Thanks to the exponential component in the divider of Eq. 19 GA tries to find a new abstract plan accepting some number of the same services which are elements of the abstract plans obtained so far. Moreover, increasing the number of the same service types included in an individual and the obtained abstract plans leads to decreasing fitness value of an individual. The value of the parameter D has been chosen experimentally and set to 1.5.

The greater the similarity between the individual and a known plan, the more the fitness value of this individual is decreased due to $D^{sim_M^{Sol}}$.

According to Eq. 19 the value of the function fit_{M,w,w_q} is calculated for each initial and expected world, denoted by w and w_q, respectively. Eventually, the maximal value of the function fit_{M,w,w_q} is set as the fitness of an individual M (see below).

$$fitness(M) = max\left\{ fit_{M,w,w_q} \mid w \in W^q_{init}, w_q \in W^q_{exp} \right\} \tag{20}$$

4.5. Mutation Operator

The mutation operator used in our approach is specialised to the problem to be solved, i.e., it makes use of the concept of good service types. A mutation of a gene m is performed only if m does not represent a good service type and if there exists a *good* service type for the sequence we consider (generated by the procedure *seqGen*). Therefore, we need first to compute a set of all service types which are good for the given sequence (see Alg. 4). If this set is nonempty, the algorithm replaces the mutated gene by its randomly selected element. The mutation algorithm presented as Alg. 5 is therefore nondeterministic, and does not work in a greedy way.

Procedure mutGST(M, q)
Input: multiset of service types: M, user query: q
Result: set of the good service types for M and q to be used by the mutation operator: GS

```
begin
    GS ← GST(M, q);
    S ← 𝕊 \ GS;
    foreach s ∈ S do
        if Tp↓(inout_s ∪ out_s) ∩ Tp↓(inout_q ∪ out_q) ≠ ∅ then
            GS ← GS ∪ {s} ; // add s to GS
            continue;
        foreach g ∈ GS do
            if Tp↓(inout_s ∪ out_s) ∩ Tp↓(in_g ∪ inout_g) ≠ ∅ then
                GS ← GS ∪ {s} ; // add s to GS
                break;
    return GS
```

Algorithm 4: Proc. *mutGST* computes a set of the good service types for a multiset and a user query to be used by the mutation operator

[2]However, the values found are obviously not universal for every ontology and every query. During the search we found, e.g., values $\alpha = 7, \beta = 1, \gamma = 2$, and $\delta = 1$ giving also very good results for benchmarks having one solution only, and slightly worse for finding many alternative plans. In our future work, we plan to search for a more general solution, e.g., to make the parameter values dependent on some measurable features of ontologies and queries.

Mutation algorithm(M, pr, q)
Input: multiset of service types: M, probability of mutation: pr, user query: q
Result: mutated individual as a new multiset of service types: M_o

begin
  $GS \leftarrow GST(M, q)$;
  $AGS \leftarrow mutGST(M, q)$;
  $M_o \leftarrow \emptyset$;
  foreach $m \in M$ **do**
    $x \leftarrow m$
    if $random(0, 1) < pr$ **then**
      if $m \notin GS$ **then**
        $x \leftarrow s \in AGS$; // choose randomly service from AGS
    $M_o \leftarrow M_o \cup \{x\}$
  return M_o

Algorithm 5: Mutation algorithm of an individual

5. Hybrid Algorithm

The third approach to APP is the hybrid algorithm, combining the SMT and the GA approach. A motivation for developing this solution comes from analysing experimental results for both the methods mentioned. The experiments, presented in Section 7, show that the main advantage of GA is a short computation time, but, concerning drawbacks, it can be observed that the longer the abstract plan the lower is the probability of finding a solution. Moreover, for the SMT-based algorithm the computation time is longer compared to GA, but all the plans are found also for instances with a greater number of services. Furthermore, only the SMT-based planner (symbolically) explores the whole state space to determine that there are no other plans limited by the given length. This check, however, is not always successful due to the timeout imposed. For large instances the SMT solver finishes computations within the time limit only when a single plan exists. If there are more plans it returns at least a single solution, but this does not guarantee that there are no more plans. Thus, both the methods have advantages and disadvantages, which leads to an obvious conclusion that the most promising approach should combine the algorithms, trying to profit from their advantages. We do this by identifying (and encoding as SMT formulas) the sub-problems of APP which are quickly solvable by an SMT-solver.

One of the main problems we have faced for some hard APP instances was that GA ends up quite frequently with unfeasible solutions containing only a few "unmatched" (i.e., not *good*) service types. The idea of our hybrid approach was therefore to solve the above problem by trying to improve, in every iteration of GA, some number of best individuals by means of the SMT-based algorithm. Thus, we replace their genes corresponding to unmatched services by appropriate good service types. Searching for these service types is performed using SMT; if they can be found, then an abstract plan is represented by the modified individual. Algorithm 6 presents the outline of the hybrid algorithm.

Going into details, our hybrid algorithm mimics the standard steps of GA, modifying them as follows: in every GA iteration,

Hybrid Algorithm(k, N, I, Cp, Mp, G)
Input: individual length: k, number of individuals in population: N, number of iterations: I, crossover probability: Cp, mutation probability: Mp, number of individuals passed to SMT: G
Result: a set of abstract plans

begin
  $P \leftarrow generateInitPop(N)$; // the initial population
  $evaluate(P)$; // compute fitness of inds. from P
  initialise SMT solver;
  $Plans \leftarrow \emptyset$;
  for $i \leftarrow 1..I$ **do**
    $T \leftarrow selection(P)$; // a temporary population
    $T \leftarrow crossover(T)$; // apply genetic operators
    $T \leftarrow mutation(T)$; // w.r.t. Cp and Mp values
    $P \leftarrow T$; // a new population obtained
    foreach $ind \in P$ **do** // evaluate population
      $evaluate(ind)$; // fitness and similarity
      if *ind is a solution* **then**
        $Plans \leftarrow Plans \cup \{ind\}$; // a new plan
    $Top \leftarrow getTopInds(P, G)$; // candidates for SMT
    foreach $ind \in Top$ **do** // check conditions
      $e_{ind} \leftarrow getExecPrefixLen(ind)$;
      $g_{ind} \leftarrow getGoodServiceNum(ind)$;
      if $(\lceil \frac{k}{2} \rceil \leq e_{ind} < k) \wedge (g_{ind} \geq \lceil \frac{k}{2} \rceil)$ **then**
        $ind' \leftarrow smtImprove(ind, Plans)$; // run SMT
        if *ind′ is a solution* **then**
          $Plans \leftarrow Plans \cup \{ind'\}$; // a new plan
          $ind \leftarrow ind'$
  stop SMT solver;
  return *Plans*

Algorithm 6: Pseudocode of the hybrid algorithm

after completing an evaluation step, some fixed number of best individuals are candidates to be improved by SMT-based algorithm. None of them represents a complete abstract plan at this stage. The individuals are selected according to the following criteria: let I denote an individual composed of k genes, $g(I)$ represent the number of good service types of I, and $e(I)$ be the length of the maximal executable prefix of I. The SMT-based procedure is called for the individual I if

- $\lceil \frac{k}{2} \rceil \leq e(I) < k$, and
- $g(I) \geq \lceil \frac{k}{2} \rceil$.

This means that each candidate for an improvement has a set of genes at least a half of which are good service types, at least a half of which constitutes an executable prefix, and which contains one or more genes to be changed. If a solution "correcting" the "bad" genes is found by SMT, the improved individual is returned to the GA population.

In order to avoid a too long computation time we use the procedure of selecting "best" individuals as well as a fixed number of individuals to be passed to the SMT procedure.

However, it should be mentioned that the criteria for "best" individuals are based on a numeric evaluation only (i.e., on comparing the number of good service types and the length of the longest executable prefix with the maximal values, which

for a k-gene individual are k). In fact, if a solution is not known, it is difficult to estimate how close an individual is from an abstract plan and whether it can be improved.

5.1. SMT-based Problem Encoding

Some technical details of our implementation of hybrid algorithm are as follows.

The symbolic SMT encoding to be combined with GA is based on the generic SMT-based method (see Section 3). However, the aim here is not just to find a solution, but to check for a given individual whether its maximal (non-executable) suffix can be modified so that the improved individual is an APP solution. Thus for the SMT procedure, the list of arguments includes not only the user query q and the ontology, but also the individual and the length of its maximal executable prefix. The role of SMT in the hybrid algorithm is shown in Figure 4.

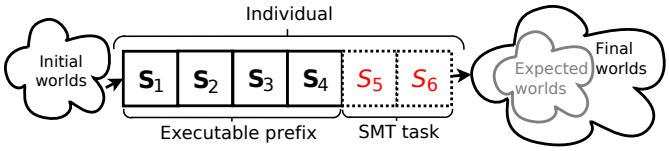


Figure 4: SMT role in the hybrid algorithm

For an individual I, let I_j be the j-th service type in I. Let k be the total length of I, and e denote the length of the maximal executable prefix. Then, the SMT encoding is represented by the following formula:

$$\varphi_k^q = \mathcal{I}^q \wedge \bigwedge_{j=1..e} \left(C_j \wedge (num(s) = I_j) \wedge \mathcal{T}_j^s\right) \wedge \bigwedge_{i=(e+1)..k} \left(C_i \wedge \bigvee_{s \in \mathbb{S}} \mathcal{T}_i^s\right) \wedge \mathcal{E}_k^q \wedge \mathcal{B}_k^q \tag{21}$$

where $\mathcal{I}^q$ and $\mathcal{E}_k^q$ encode the initial and the expected worldset of the user query, respectively, C_i encodes the i-th context function, and $\mathcal{T}_i^s$ encodes the worlds transformation by a service type s. $\mathcal{B}_k^q$ blocks the solutions found so far. Thus, the SMT-solver searches for a sequence of at most $(k - e)$ service types, replacing the non-executable suffix of I so that the improved individual is an (unknown so far) solution of APP.

6. Pruning ontologies using a graph database

Typically, an ontology is very large while user queries tend to be local, producing plans containing small numbers of services and objects, compared to the overall ontology size. The Planics planners developed so far worked on whole ontologies what hindered their performance and left a lot of space for improvements. In this section we show that the abstract planning problem (restricted to matching types of services) can be translated to the reachability problem for graphs. Then we present an approach of using a graph database to pruning ontologies for given user queries.

6.1. Graphs

Standard definitions of graphs and graph related notions are used in this section (see part IV of [19]). We start with introducing subgraphs induced by paths and graph databases. A set of paths S_P in a graph G induces the subgraph G' such that each vertex and each transition of G' is an element of some path of S_P. This intuition is captured by the following definition:

Definition 16 (Subgraph induced by paths)**.** *Let S_P be a set of paths of a graph $G = (V, E)$. The subgraph $G' = (V', E')$ of G induced by S_P is defined as follows:*

- $V' = \{v \in V \mid \exists p \in S_P : v \in p\}$ *and*
- $E' = \{(v, v') \in E \mid \exists p \in S_p : (v, v') \in p\}$.

6.2. Graph Databases

A *graph database* is a tool for storing directed graphs and finding their subgraphs satisfying certain properties, defined over vertices and edges that are expressed by database queries[3]. From a formal point of view, this is a graph algorithm with a precisely defined semantics. For example, one could map a group of persons to vertices, friendship relations between these persons to edges, and formulate a database query to find everyone who is female, older than 50 years (assuming that every vertex has an attribute for the age of the person it represents) and has at least two friends shared with another person.

For some applications, it is more convenient to consider a variant of the reachability problem of finding all the paths between two explicitly known sets of vertices: the initial and the final set. This can be easily reduced to finding all the paths between a pair of vertices, one with an outcoming edge to every element of the initial set, and the other one with an incoming edge from every vertex of the final set. The latter approach is conceptually simpler and sometimes more efficient. Since it can be applied to pruning ontologies, we describe it formally. To this aim, we begin with showing how a given graph is stored in the database, where the operations *addNode*() and *addEdge*(), having a clear meaning related to their names, are applied.

Definition 17 (Graph database)**.** *Given a directed graph $G = (V, E)$. G is said to be* stored *in a graph database DB if addNode(v) is executed for every vertex $v \in V$ and addEdge(v, v') is executed for every edge $(v, v') \in E$.*

Then, searching a stored graph is defined.

Definition 18 (Graph database query result)**.** *Let $G = (V, E)$ be a directed graph stored in a graph database DB. Let $q_{DB}^k = (v_I, v_F, k)$ be a database query, where $v_I, v_F \in V$ and $k \in \mathbb{N}$. The result of the query q_{DB}^k applied to DB is the subgraph $G_q \subseteq G$, induced by the set of paths of G of length at most k, which begin with v_I and end with v_F. We refer to G_q by result(G, q_{DB}^k).*

[3] One should not confuse a database query (specifying what is to be found in the database) and a user query (specifying the task of the composition process).

From a practical perspective, graph databases can be used for storing and search effectively very big state spaces. In addition, the graph representation enables for a graphical visualization of the graphs stored in the database as well as the query results. Advanced features are available in the area of guaranteeing data redundancy, distributing data between multiple machines, and optimizing the database performance.

6.3. Pruning an Ontology

Our graph-based approach to pruning ontologies for abstract planning consists of the following stages:

1. Choosing an upper bound (k) on the plans length for which the search is to be performed.
2. Encoding the ontology in the graph database. Every object type and service type of the ontology is represented by a distinguished vertex. The edges connect pairs of vertices, where one vertex models a service type while another one an object type. For a vertex representing a service type s, an incoming edge from a vertex representing an object type models that this object type is an input for the service type s. Similarly, an outgoing edge to a vertex representing an object type models that this object type is an output of the service type s. The rules are also applied to the object types derived from every object type occurring in the input and output lists of a service type. After application of these rules we get the *ontology graph* G_{Ont}.
3. Extending the ontology graph G_{Ont} to the *query graph* G_q, where $G_{Ont} \subseteq G_q$, by adding the *initial* vertex and the *final* vertex, for a user query q. The outgoing edges of the initial vertex are connected to the vertices representing the object types of the initial worldset of q and to the vertices representing the service types having empty input lists. The edges ingoing to the final vertex start from the vertices representing the object types and subtypes of the expected worldset of q.
4. Searching for a set of the paths (of length restricted to k) between the initial and the final vertex of the query graph G_q. This objective is expressed by a database query fed to the graph database. The result is the *query subgraph* $G_{qs} \subseteq G_q$.
5. (Optional Postprocessing) Removing recursively from the query subgraph G_{qs} the vertices representing service types, for which some object types of their input lists have not been identified by the search. Removing them is not necessary, because we will add missing types when building the pruned ontology. However, those service types cannot be executed in the reduced ontology, so filtering them out makes it smaller and, hopefully, easier to handle for the planners. We can also remove the vertices of object types not connected to any vertices representing service types.
6. Pruning the original ontology to the service and objects types represented by the vertices of the query subgraph G_{qs}. Then, adding the types of the objects possibly produced by the service types in the reduced ontology and not relevant for any abstract plan which can be found, but required for an ontology to be complete. The pruned ontology can replace the full ontology in the planning process.

The algorithm described above can be repeated for an incremented depth k or run for any depth. It is complete and sound for the minimal abstract plans of length restricted by the depth k chosen as a parameter. This means that the ontology pruned preserves all the minimal abstract plans of length k and does not introduce any abstract plans which could not be generated starting with the original ontology.

Now we are in a position to describe the algorithm formally.

6.3.1. Ontology Graph: Encoding an Ontology

For the purpose of this section we recall some notions related to Planics, which are used in the formalism below.

Definition 19 (Ontology)**.** *By an* ontology *we mean a triple* $Ont = (\mathbb{S}, \mathbb{T}, Ext)$*, where*

- $\mathbb{S}$ *is the set of all the service types,*
- $\mathbb{T}$ *is the set of all the object types, i.e., the types of* $Artifact$ *and* $Stamp$*, and their descendants,*
- Ext *is the inheritance relation of the object types.*

Moreover, we recall the function $\mathcal{T}p^{\downarrow} : 2^{\mathbb{O}} \longmapsto 2^{\mathbb{T}}$, such that $\mathcal{T}p^{\downarrow}(O) = \bigcup_{o \in O}\{t \in \mathbb{T} \mid t = \mathcal{T}p(o) \vee (\mathcal{T}p(o), t) \in Ext\}$ which assigns the set of the types and subtypes to each set of objects.

Assume we are given an ontology Ont. Our first task is to encode Ont as a graph. It is quite common to use graphs for modeling the inheritance of classes (see Fig. 1). We extend this approach by modeling service and object types as vertices of a graph stored in the graph database, and encoding with the edges the relation of processing and producing the objects by service types. In particular, we introduce the directed edges connecting the vertices representing the service types with the vertices corresponding to the (sub)types of the objects processed, in the way captured by the following definition.

Definition 20 (Ontology graph)**.** *Given an ontology Ont. By the* ontology graph *we mean the graph* $G_{Ont} = (V_{Ont}, E_{Ont})$*, where*

- $V_{Ont} = V_{\mathbb{S}} \cup V_{\mathbb{T}}$ *with* $V_{\mathbb{S}} = \{v_s \mid s \in \mathbb{S}\}$ *and* $V_{\mathbb{T}} = \{v_{\boldsymbol{t}} \mid \boldsymbol{t} \in \mathbb{T}\}$*,*
- $E_{Ont} = E_{\mathbb{S}} \cup E_{\mathbb{T}}$ *with*

  $E_{\mathbb{S}} = \{(v_s, v_{\boldsymbol{t}}) \mid s \in \mathbb{S} \ \wedge \boldsymbol{t} \in \mathcal{T}p^{\downarrow}(out_s \cup inout_s)\}$,
  $E_{\mathbb{T}} = \{(v_{\boldsymbol{t}}, v_s) \mid s \in \mathbb{S} \ \wedge \boldsymbol{t} \in \mathcal{T}p^{\downarrow}(in_s \cup inout_s)\}$.

Example 4. *Ontology graph. In Fig. 5 we show an example of the ontology graph for object types introduced in Section 2 extended with the type* $Certificate$ *(derived from* $Artifact$*). Moreover, the service type* $Inspection$ *has been introduced, which is defined as follows:*

$spec_{Inspection} = (\ \text{in} = \{(a, Arbour)\}, \text{inout} = \emptyset,$
$\text{out} = \{(c, Certificate), (stamp, PriceStamp)\},$
$\text{pre} = \texttt{isSet}(a.id)\ \texttt{and}\ \texttt{isSet}(a.owner)\texttt{and}\ \texttt{isSet}(a.location),$
$\text{post} = \texttt{isSet}(c.owner)\ \texttt{and}\ \texttt{isSet}(stamp.price)).$

The rectangles correspond to the object types, the ovals represent service types. The dashed arrows model the ontology types hierarchy, while solid arrows correspond to the edges of the ontology graph. Note, that only black nodes and arrows are members of the ontology graph. For better readability we show here only two service types.

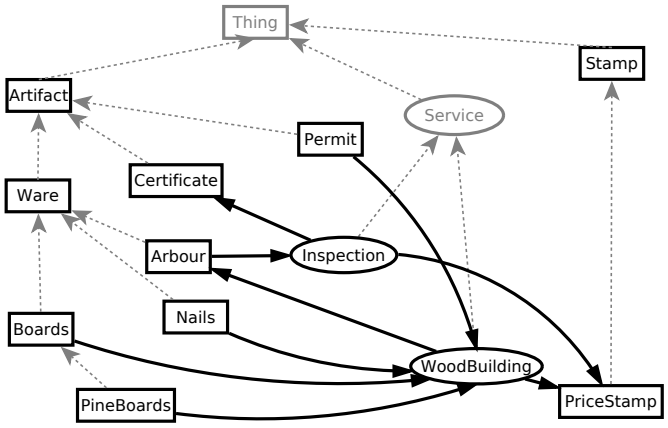


Figure 5: A fragment of example ontology graph.

6.3.2. Query Graph

The ontology graph G_{Ont} (stored in the graph database) represents the ontology Ont. The next step is to extend G_{Ont} with a representation of a user query q by constructing a *query graph*, which contains also the *initial vertex* v_I and the *final vertex* v_F. The initial vertex is connected to the vertices corresponding to the types of the objects from the initial worldset. The vertices corresponding to the types and subtypes of the objects of the expected worldset of the user query are connected to the final vertex. Notice that v_I is also connected to the vertices corresponding to each service type having the list *in* and *inout* empty.

For the following definition of a query graph, we need an extension of the function $\mathcal{T}p$ to sets of objects, i.e., the function $\mathcal{T}p$ assigns a set of the types to all object of each set $O \subseteq \mathbb{O}$. Formally: $\mathcal{T}p : 2^{\mathbb{O}} \longmapsto 2^{\mathbb{T}}$ such that

- $\mathcal{T}p(O) = \{t \in \mathbb{T} \mid \exists o \in O : t = \mathcal{T}p(o)\}$,

Definition 21 (Query graph)**.** *Given the ontology graph* $G_{Ont} = (V_{Ont}, E_{Ont})$ *and a user query specification* $q = (in_q, inout_q, out_q, pre_q, post_q)$*, where* $inout_q \cup out_q \neq \emptyset$[4]*. By the* query graph *we mean the graph* $G_q = (V_q, E_q)$*, where:*

- $V_q = V_{Ont} \cup \{v_I, v_F\}$*, where* $V_{Ont} \cap \{v_I, v_F\} = \emptyset$*,*
- $E_q = E_{Ont} \cup \{(v_I, v_t) \mid v_t \in V_{\mathbb{T}} \wedge \ (\boldsymbol{t} \in \mathcal{T}p(in_q \cup inout_q)\} \cup$
  $\{(v_I, v_s) \mid v_s \in V_{\mathbb{S}} \wedge \ (in_s \cup inout_s = \emptyset)\} \cup$
  $\{(v_t, v_F) \mid v_t \in V_{\mathbb{T}} \wedge \ (\boldsymbol{t} \in \mathcal{T}p^{\downarrow}(out_q \cup inout_q))\}$.

Notice that the subtypes are added only for the object types of the expected worldset which corresponds to the fact the the user accepts 'more' than he requires. Clearly, one cannot assume that the user possesses 'more' than specified by the initial worldset, so no subtypes of the initial worldset objects are introduced.

[4]For the empty $inout_q \cup out_q$ there is no need to search for a solution, so to reduce an ontology.

6.3.3. Query k-subgraph: Pruning Query Graph

Our next step consists in pruning the query graph leaving only its subgraph (called the *query k-subgraph*) induced by all the paths of length k from the initial vertex to the final one. This subgraph is produced as the result of a database query to the graph database storing the query graph. In this query the depth is given by $2k + 2$ to reflect the fact that a solution of length k corresponds to a path of length $2k + 2$ in the query graph because of its construction.

Below we formalize the above concept.

Definition 22 (Query k-subgraph)**.** *Let* $G_q = (V_q, E_q)$ *be the query graph and* $k \in \mathbb{N}$*. The* query k-subgraph $G^k_{qs} \subseteq G_q$ *is the result of executing the database query* $Q^{2k+2} = (v_I, v_F, 2k + 2)$ *onto the query graph, where* v_I, v_F *are the initial and final vertices of* G_q*, respectively.*

In order to define formally the ontology pruned we need the notion of *supertypes* of a set object types. Formally, for $T \subseteq \mathbb{T}$ we define $\mathcal{T}p^{\uparrow}(T) = \bigcup_{t \in T}\{\boldsymbol{t'} \in \mathbb{T} \mid (\boldsymbol{t'}, \boldsymbol{t}) \in Ext^*\}$.

The pruned ontology is the final result of the graph reduction. As a special case, the empty sets of service and object types are returned if there exists an object type of the expected world which cannot be produced (i.e., there is no path in the query subgraph leading to the node modeling it, and the same holds for all its subtypes).

We say that Ont_k is an *empty ontology* if it has empty sets of services and object types. Sometimes, empty ontologies can be identified by a simple property of the query k-subgraph:

Definition 23 (query k-subgraph generating empty ontology)**.** *Let* $G_{qs} = (V_{qs}, E_{qs})$ *be the query k-subgraph. We say that* G_{qs} *generates the empty ontology if* $\exists o \in out_q$ *(so* $\mathcal{T}p(o)$ *is represented in* G_{qs}*) and for* $\forall \boldsymbol{t} \in \mathcal{T}p^{\downarrow}(\{o\})$*, there are no incoming transitions to* $v_{\boldsymbol{t}} \in V_{qs}$*.*

The pruned ontology is formally defined as follows:

Definition 24 (k-Reduced ontology)**.** *Let* $Ont = (\mathbb{S}, \mathbb{T}, Ext)$ *be an ontology and* $G_{qs} = (V_{qs}, E_{qs})$ *be the query k-subgraph. By the* k-reduced ontology *we mean the ontology* $Ont_k = (\mathbb{S}_k, \mathbb{T}_k, Ext_k)$*, which is empty iff* G_{qs} *generates the empty ontology, and otherwise defined as follows:*

- $\mathbb{S}_k = \{s \in \mathbb{S} \mid v_s \in V_{qs}\}$,
- $\mathbb{T}_k = \mathcal{T}p^{\uparrow}\Big(in_q \cup inout_q \cup \bigcup_{s \in \mathbb{S}_k} \mathcal{T}p^{\downarrow}(in_s \cup inout_s \cup out_s)\Big)$,
- $Ext_k = Ext \cap (\mathbb{T}_k \times \mathbb{T}_k)$.

The k-reduced ontology contains all the service types and all the object types corresponding to the vertices of G^k_{qs}. In addition, it contains all the supertypes of the subtypes of the sets of the input and output object types of each its service type. An example fragment of a query subgraph is shown in Fig. 6.

Note that the subtypes of out_q, with their supertypes, are already present in the pruned ontology, since otherwise the ontology would be empty. However, the objects types of $in_q \cup inout_q$ are explicitly added to the reduced ontology since there could

be a solution which does not use these types, but the reduced ontology needs to be complete.

In Definition 24, the motivation for adding the supertypes of the object types from service argument lists is the same as for types from the user query, i.e., so that the type system in the ontology pruned should be complete.

6.4. Correctness of the Reduction

Now we prove that the reduction of the ontology preserves all the *minimal* user query solutions, i.e., intuitively, such solutions which contain only service types necessary for satisfying the user query. In order to prove the correctness, for each solution we define a set of *object type derivation sequences (OTDSs)*, which are subsequences of the solution (without contexts), in which all two consecutive service types share an object type between their inputs and outputs. We show that each service type s of a minimal solution is present in some OTDS. Finally, we show that each OTDS can be mapped to a path in the query k-graph, so all the object and service types needed to preserve the minimal user query solutions are present in the reduced ontology.

Definition 25 (Object type derivation sequence (OTDS))**.** *Given an ontology Ont, a user query q, a q-solution* seq $= ((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k}))$ *of length k, and* $m \leq k$*. A* quasi object type derivation sequence (QOTDS for short) $qotds = (s_{i_1}, s_{i_2}, \ldots, s_{i_m})$ *is a subsequence of* $(s_1, \ldots, s_k)$*, satisfying the following two conditions:*

1. $\mathcal{T}p(in_q \cup inout_q) \cap \mathcal{T}p^{\downarrow}(in_{s_{i_1}} \cup inout_{s_{i_1}}) \neq \emptyset$, *if* $in_{s_{i_1}} \cup inout_{s_{i_1}} \neq \emptyset$,
2. $\mathcal{T}p^{\downarrow}(out_{s_{i_j}} \cup inout_{s_{i_j}}) \cap \mathcal{T}p^{\downarrow}(in_{s_{i_{j+1}}} \cup inout_{s_{i_{j+1}}}) \neq \emptyset$, *for* $1 \leq j \leq m-1$.

A QOTDS qotds is called an object type derivation sequence (OTDS for short) *if qotds satisfies also the following condition:*

3. $\mathcal{T}p^{\downarrow}(out_{s_{i_m}} \cup inout_{s_{i_m}}) \cap \mathcal{T}p^{\downarrow}(inout_q \cup out_q) \neq \emptyset$.

Note that the definition of OTDS is not using context functions of *seq* as we are working only at the level of matching object types without referring to their actual values. It is easy to show that for each q-solution there is at least one OTDS, but there could be many of them of different lengths.

We write $(Q)OTDS(seq))$ for all the (Q)OTDSs for a q-solution *seq* for a given implicitly ontology *Ont*.

Our ontology pruning method preserves the *minimal* q-solutions only, where every service type is necessary for satisfying q. This notion is captured by the following definition:

Definition 26. *[Minimal solution] Let* seq $= ((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k}))$ *be a q-solution. We say that* seq *is a* minimal solution*, if no strict subsequence of* seq *is a q-solution.*

It is easy to show that for each q-solution, there is always a corresponding minimal q-solution

Lemma 1. *Let* seq $= ((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k}))$ *be a q-solution. There is a subsequence* seq′ *of* seq *which is a minimal q-solution.*

Proof. If *seq* is not minimal, then by the above definition we can remove from *seq* elements until we reach *seq′* which is a minimal q-solution. □

Notice that a q-solution may contain many minimal q-solutions.

Lemma 2 (Characterisation of OTDSs)**.** *Let* seq $= ((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k}))$ *be a q-solution. Consider any* s_i *with* $1 \leq i \leq k$*. If* $s_i \notin otds$ *for each* $otds \in OTDS(\text{seq})$*, then for some c:* $i \leq c \leq k$*,* $(s_c, ctx_{O_c}^{s_c})$ *can be removed from* seq *and the resulting sequence* seq′ *is still a q-solution.*

Proof. See part II of [19]. □

The next lemma states that OTDSs of $OTDS(seq)$ contain the service types of every minimal solution *seq*.

Lemma 3. *Let* seq $= ((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k}))$ *be a minimal q-solution, for* $k \in \mathbb{N}$*. Then, for every* $1 \leq i \leq k$*,* s_i *belongs to some* $otds \in OTDS(\text{seq})$.

Proof. Let *seq* be a minimal q-solution and suppose that for some $1 \leq i \leq k$ s_i is not an element of any $otds \in OTDS(seq)$. So, by Lemma 2 we know that for some $c \geq i$, $(s_c, ctx_{O_c}^{s_c})$ can be removed from *seq* and the resulting sequence *seq′* is still a q-solution, which is a contradiction to the fact that *seq* is minimal. □

Note that in general, there are more than one otds for each solution. Now we show that every element of $OTDS(seq)$ is represented by a path in the query k-subgraph.

Lemma 4. *Let seq be a q-solution. For each* $otds = (s_{i_1}, s_{i_2}, \ldots, s_{i_m}) \in OTDS(\text{seq})$ *there exists a path p in* G_{qs}^k*, where* $m \leq k$*, such that p contains a vertex* $v_{s_{i_j}}$ *for all* $1 \leq j \leq m$*, i.e., all the service types of otds are represented in* G_{qs}^k*.*

Proof. See part II of [19]. □

Note that if follows from the above lemmas that the reduction depth equal to the length of the longest OTDS is sufficient to preserve all minimal plans. This observation enables further optimisations by iteratively increasing the reduction depth, especially when we deal with huge ontologies and the priority is to find a plan, but not necessarily all of them have to be preserved.

Example 5. *Assume the ontology from Example 4 and the query* $(in_q = \{(P, Permit)\}$*,* $inout_q = \{(N, Nails), (B, Boards), \}$*,* $pre_q = true$*,* $out_q = \{(A, Arbour), (St, PriceStamp)\}$*,* $post_q =$ `isSet`$(A.owner))$*. Fig. 6 illustrates the reduction on depth* 1 *(the white nodes), and* 2 *(the white and gray nodes). The triangle nodes labelled with S and F correspond to the initial and final nodes, respectively. It is easy to see that the reduction of depth* 1 *is enough for finding a solution, since we consider all the paths from S to F of maximal length* $2*1+2 = 4$*. Increasing the reduction depth introduces additional types (Certificate, Inspection) which are not necessary for the plan.*

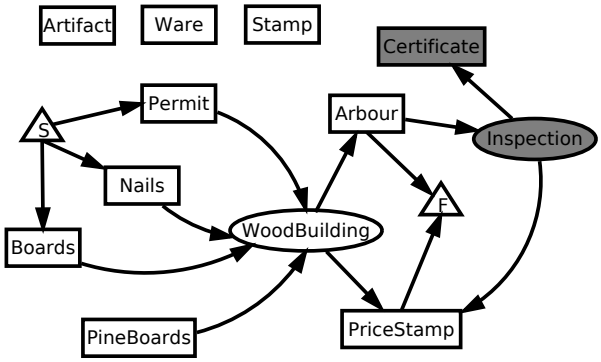


Figure 6: A fragment of query ontology graph.

It can be easily seen that the presented abstraction is in fact an over-approximation. Some of the paths in the query k-subgraph returned by the database can have no corresponding object type derivation sequences for any solution.

When, for every expected world, there is no path to at least one of its object types in the query subgraph, this means that no valid solution exists. However, the existence of such paths for all the object types of an expected world does not guarantee that there is a solution. This is because the graph approach works at the level of types, and does not take into account the issues such as checking pre- and postconditions, and providing enough objects for cardinality constraints. Checking the pruned ontology by an abstract planner is still needed, but usually the scope of this search is significantly reduced. Thus, the method is complete and sound, and every valid plan will be found in the query subgraph for the chosen depth.

6.4.1. Implementing the Algorithm in the Graph Database

As we have described above, the graph database is to represent the ontology graph and, for every user query, to extend it to the query graph to find the query subgraph. We have used the graph database Neo4j, working in the standalone mode (that is, without the server installation, but run over a Java API communicating with the database, performing operations such as adding vertices and edges, and labeling them). The graphs are directly represented by the graph database following their semantical model, without any transformations. The general way of interaction with the database is as follows: first the ontology graph is stored in the database. It can be expected to be of a significant size, but its construction is performed only once, independently of the user queries. Then, for each user query, the query graph is constructed by adding the initial and final vertex, with respective edges. Next, the database query expressed in the *Cypher* language is passed to Neo4j. The database query is the same regardless of the user query, and has the following meaning: *find all the paths of depth at most* $2k+2$*, between the start and the final vertices*. Then, the resulting query subgraph is a basis for construction of the pruned ontology. Finally, the start and final vertices with their edges are removed from the database, and the system is ready for the next user query.

7. Experimental Results

We have implemented all our planners and evaluated their efficiency using the ontologies and the user queries produced by the customizable Ontology Generator (OG). The benchmarks are parametrized, thus convenient for checking scalability of our planners in finding plans. This is achieved by fixing different values of several parameters and characterizing the exemplary ontologies. We continue with a short description of OG.

7.1. Ontology Generator

OG produces ontologies and user queries in accordance with its input parameters and the semantics rules. Also the corresponding user queries are provided, guaranteeing the requested number and format of solutions.

Parameter	Explanation – number of:	Default value
$n_{\|\mathbb{O}\|}$	object types	100
n_{OA}^{min}	the object attributes (minimal)	1
n_{OA}^{max}	the object attributes (maximal)	2
$n_{\|\mathbb{S}\|}$	service types in ontology	64
n_{OinS}^{min}	objects in service type input lists (minimal)	
n_{OinS}^{max}	objects in service type input lists (maximal)	
$n_{\|EW\|}$	the objects required by a user in the expected world (thus branches)	
len	service types in every branch (both extended and not extended)	
n_{ext}	extended branches	0
$n_{\|ext\|}$	independent sub-branches in every extended branch (including the original single sub-branch)	0

Table 1: Meaning of the parameters characterizing benchmarks.

Table 1 describes the parameters characterizing the benchmarks. The parameters listed above the bold line were used in some previous papers about Planics [11, 20, 12, 21]. However, the benchmarks generated using these parameters appear to have a very specific structure, which, in many cases, may facilitate finding plans. Namely, for every plan, each object of the expected worldset is produced by a sequence of services (called a *branch*), which is independent of all the other services in the plan. By being independent we mean that the services belonging to a branch do share neither produced nor consumed objects with any other services, possibly with the exception of some objects of the initial worldset. Thus, we have modified the ontology generator introducing additional parameters in order to obtain benchmarks closer to real-life examples.

7.1.1. Extending Branches

First, OG is modified in order to provide more ways to produce objects of an expected world. Now, we can easily increase the number of plans by manipulating the values of the two parameters: n_{ext} and $n_{|ext|}$, where $0 \leq n_{ext} \leq n_{|EW|}$. Thus, every object can be produced not only by a single branch, but also by $n_{|ext|}$ *sub-branches* for every branch extended in this way. By an *extended branch* we mean all the sub-branches producing a single object of an expected world. Every sub-branch is independent of all other sub-branches and branches. This modification allows to check how the planners deal with a large number of plans.

7.2. Configuration of the Experiments

The experiments have been conducted on a virtual machine with assigned 4 CPUs and 16 GB RAM running Ubuntu Linux deployed on a server equipped with two 2.40 GHz processors Intel Xeon E5-2630v3. The version 4.4.1 of Z3 [14] has been used as an SMT-solving engine. The GA and Hybrid planners have been run 30 times for each benchmark, while SMT three times[5] in order to report the average times at this basis. We have imposed the 2000 sec. CPU time limit for every experiment. The memory usage remains below 2 GB even for the "pure SMT" planning on the largest instances. The memory consumption by the SMT-solver increases at the end of the search, i.e., while determining that there no more plans exist. The memory usage for the GA algorithm is usually below several hundred MBs, and the hybrid algorithm does not need more than 1 GB, even for the largest instances.

7.3. Evaluation of Experimental Results

In this section we give descriptions of the experiments, benchmark definitions, and the corresponding results[6]. We have performed several experiments in order to investigate how selected features affect the planning efficiency. Note that some features might be influenced by more than one parameter. Thus, in every experiment we scale these parameters which influence the observed benchmark features. Since each planning method has advantages and drawbacks that become apparent in some specific situations, we divide each experiment into several benchmark groups corresponding to low, medium, and high values of the scaled parameters[7]. Another reason for the benchmark grouping is their large number. We have run over seventy benchmarks, but their clustering facilitates the overall results analysis as well.

For every experiment we report the results of applying all our planners to complete and pruned ontologies. We evaluate the computation time, the number of plans found, and the probability of finding a solution. Moreover, in order to compare efficiency of our planning algorithms with other tools, we present also the results of the Fast Downward (FD) tool [18] applied to the same benchmarks. Since FD is aimed at solving problems specified in PDDL [53], we have translated our ontologies and queries into PDDL. More details on the translation is given in Section 7.4.

We start with investigating how the number of plans influences the planning efficiency.

7.3.1. Experiment 1: scaling the number of plans

The two parameters of the modified OG affect the number of plans: $n_{|ext|}$ and n_{ext}. In Experiment 1.1, we scale the parameter $n_{|ext|}$, while Experiment 1.2 and 1.3 show the impact of n_{ext} on the planners performance. Table 2 presents the parameters and features of Experiment 1.1 where $n_{|ext|}$ ranges from 2 to 10 and all the other parameters remain constant. The double column separators indicate the division of the benchmarks into three groups. The two last rows of the table show the number of service and object types remaining after the GDB reduction. As one should expect, the more plans, the more types have to be kept in order to preserve all minimal solutions. All benchmarks from this group yield from 4 to 100 abstract plans of length 6.

Table 3 presents the planning results on complete ontologies. The columns from left to right display the benchmark id, the performance of the GA planner described by the probability of finding a solution, the average and maximal number of plans found, and the computation time. Next, we give the results of the SMT-based planning. That is, the number of plans found, the time of finding the first and all the other plans, the time of checking whether more plans exist, and the total computation time. The next columns show the performance of the hybrid planer in terms of the probability, the average and maximal number of plans found, the time consumed by the SMT and GA parts of the algorithm, and the total computation time. Finally, we show the results of the FD tool, i.e., the number of plans found, the computation time, and the memory consumption. Note that FD is aimed at finding only one plan, thus the number of plans found is either 1 or 0. We show also the memory consumption of FD, because, contrary to our algorithms, in some cases FD consumes all the memory available, and the computation fails. Note that by the computation time we mean solely the consumed CPU time.

Concerning the planers' time consumption, it is easy to observe that these are almost constant in the case of GA (about 4 sec.) and the hybrid planner (about 7 sec.). These facts match our expectations, because the GA runtime depends mostly on the length of the plan, which is constant here. The time consumption of the hybrid planner is determined mostly by the SMT component runtime. Since in this benchmark group the tasks for the solver are quite easy (there is only a few genes to improve), the SMT-based procedure does not consume more time than the GA part, and the overall runtime is satisfactory. In the case of the SMT-based planner, the total computation time grows together with the number of plans. This is a consequence of two factors. Firstly, decoding of a plan takes some time due

[5] The results returned by the SMT-based planner are deterministic and "stable", i.e., since the differences of CPU time consumption are negligible, a few repetitions of each experiment is sufficient.

[6] All the resources needed to reconstruct the experiments, i.e., the tools and the benchmark files, are available for download from http://planics.uph.edu.pl/asoc/asoc.zip

[7] Or low and high values only, depending on the number of benchmarks in each experiment.

Table 2: Experiment 1.1: the parameters and properties of the benchmarks.

	$k = 6, len = 2, n_{\|EW\|} = 3, n_{ext} = 2, \|\mathbb{O}\| = 102, \|\mathbb{S}\| = 64, sp = 2^{36}$								
id	**1.11**	**1.12**	**1.13**	**1.14**	**1.15**	**1.16**	**1.17**	**1.18**	**1.19**
$n_{\|ext\|}$	2	3	4	5	6	7	8	9	10
$\|CAP\|$	4	9	16	25	36	49	64	81	100
$\|\mathbb{S}_{pr}\|$	10	14	18	22	26	30	34	38	42
$\|\mathbb{O}_{pr}\|$	12	15	18	21	24	27	30	32	35
sp_{pr}	2^{20}	2^{23}	2^{25}	2^{27}	2^{28}	2^{29}	2^{31}	2^{31}	2^{32}

Table 3: The results of Experiment 1.1 for the unreduced ontologies

	GA			**SMT**					**Hybrid**					**FD**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
1.11	100	3.5/4	3.9	4	3.9	1.6	3.2	8.7	100	3.7/4	3.2	3.8	7.0	1	15.02	320
1.12		4.8/9	3.8	9	4.3	1.5	6.9	12.7		7.2/9	3.7	3.9	7.6		4.89	160
1.13		6.2/13	3.9	16	4.1	4.2	5.8	14.1		9.2/14	3.4	3.8	7.2		6.16	183
1.14	90	3.1/12	4.1	25	4.7	4.9	9.1	18.7		8.1/21	3.6	4.1	7.6		6.72	191
1.15	77	2.1/9	4.2	36	3.8	4.7	9.3	17.9		8.6/25	3.7	4.2	7.9		8.41	216
1.16	80	2.3/6	6.2	49	3.1	7.1	8.8	19.1		7.3/18	3.2	4.1	7.3		10.41	256
1.17	77	1.8/5	4.1	64	2.3	7.7	12.6	22.6	97	8.4/36	2.7	4.1	6.8		13.5	301
1.18	70	2.2/5	4.1	81	2.0	11.1	18.5	31.6	100	10.0/38	2.8	4.0	6.8		15.44	382
1.19		2.3/5	4.1	100	1.7	11.2	25.6	38.6		10.5/27	3.0	4.1	7.1		18.67	380

Table 4: The results of the Experiment 1.1 for the pruned ontologies. The probability of finding a plan by the Hybrid planner equals 100%.

	DB	**GA-pruned**			**SMT-pruned**					**Hybrid-pruned**				**FD-pruned**		
id	**time** [s]	**Pr. [%]**	**plans. avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
1.11	1.6	100	4.0/4	2.8	4	0.6	1.0	1.0	2.6	3.9/4	0.5	2.9	3.4	1	0.11	71
1.12	1.5		6.0/9	3.4	9	0.8	1.9	2.5	5.3	7.9/9	0.8	3.0	3.8		0.39	73
1.13	1.9		6.7/12	2.9	16	1.2	2.7	4.3	8.2	10.5/16	1.0	3.1	4.1		0.68	81
1.14	1.5		3.6/9	3.2	25	1.5	3.5	5.4	10.4	10.9/21	1.2	3.6	4.8		1.57	99
1.15	1.5		2.8/7	3.2	36	1.0	4.7	5.3	11.1	11.1/21	1.3	3.5	4.8		2.46	118
1.16	1.6	83	3.5/8	5.0	49	1.2	5.8	6.4	13.5	10.4/20	1.5	3.5	5.0		3.89	145
1.17	1.6	90	3.3/10	3.4	64	1.6	6.2	9.0	16.8	12.1/28	1.7	3.6	5.3		5.7	179
1.18	1.5	87	2.8/8	5.2	81	2.3	11.7	10.9	24.9	11.8/27	1.9	3.6	5.5		7.87	213
1.19	1.6	70	3.0/8	3.5	100	1.7	9.0	14.7	25.5	14.5/58	2.0	3.7	5.7		10.81	264

to additional interactions with the solver. Secondly, after finding a plan the blocking formula is extended and thus (usually) becomes harder to solve.

Concerning the probability of finding a plan, only the GA-based planner is affected with the growing number of solutions: one can observe that the probability decreases with the increase of the number of existing plans. This is caused by the fact that in the presence of a large number of existing plans, the solutions being "a mix" of two (or more) slightly different plans are assessed high, but sometimes it is hard for GA to direct the search into one of them. In such situations GA ends up with many individuals having only one mismatched gene, but not with a complete solution. This drawback is offset by the SMT-based component of the hybrid algorithm, and for this group of the benchmarks the hybrid planner is able to find plans with almost 100% probability, with just one little exception. The SMT-based planner always finds a plan.

As to the number of plans found, there is no surprise that the SMT-based planner finds all of them, since the computation times are all far below the imposed time limit. On the other hand, it is interesting to notice that the (average, as well as maximal) number of plans found by GA initially increases, but then decreases significantly. The explanation is the measure of similarity used to compute the fitness function which lowers the assessment of the individuals which only slightly differ from some plan found. The more plans exist, the more similarity between them. Thus, from some point GA is unable to find more than several plans. Again, the hybrid algorithm shows its superiority over GA in this area, because the number of plans found by the hybrid is much higher, and there is no such a downward trend like in the case of GA.

Concerning the results yielded by FD, in general both the consumed time and memory grows with increasing the number of possible plans. However, FD finds a plan for each benchmark, the runtimes are acceptable, and the consumed memory amount stays below 400MB.

Now, we let's summarize the results of Experiment 1.1 for the pruned ontologies of Table 4. The general conclusion is evident. The GDB substantially reduces the search space, which improves all the assessed features of the algorithms like time, probability, and the number of solutions found, in comparison to dealing with the complete ontologies. Moreover, most of the trends identified before still holds after the pruning has taken place. The same, but to an even greater extent, holds for the FD results: the consumed time and memory drops significantly after the ontology reduction. The GBD reduction time is ac-

Table 5: Experiment 1.2 (left) and 1.3 (right): the parameters and properties of the benchmarks for $n_{|ext|} = 2$, $|\mathbb{O}| = 102$, and $|\mathbb{S}| = 64$.

	$len = 2, n_{\|EW\|} = 5, k = 10, sp = 2^{60}$				
id	**1.21**	**1.22**	**1.23**	**1.24**	**1.25**
n_{ext}	1	2	3	4	5
$\|CAP\|$	2	4	8	16	32
$\|\mathbb{S}_{pr}\|$	12	14	16	18	20
$\|\mathbb{O}_{pr}\|$	17	18	19	21	23
sp_{pr}	2^{36}	2^{38}	2^{40}	2^{42}	2^{43}

$len = 3, n_{\|EW\|} = 4, k = 12, sp = 2^{72}$				
1.31	**1.32**	**1.33**	**1.34**	**1.35**
1	2	3	4	5
2	4	8	16	32
15	18	24	30	30
20	22	28	32	35
2^{47}	2^{50}	2^{55}	2^{59}	2^{59}

Table 6: The results of Experiment 1.2 and 1.3 for complete ontologies

	GA			**SMT**					**Hybrid**					**FD**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
1.21	100	1.8/2	7.0	2	26.4	4.1	1296	1326.5	100	1.8/2	17.0	7.3	24.3		5.29	169
1.22	90	2.1/4	7.1	4	24.0	13.5	748.3	785.8		3.1/4	16.8	7.6	24.5		10.43	257
1.23	93	3.0/5	6.6	8	24.0	15.8	1571	1610.8	97	4.1/7	18.5	7.2	25.7	1	14.13	311
1.24		3.4/6	6.7	16	25.3	29.8	tmOut	tmOut	100	6.2/13	20.3	7.2	27.4		4.97	163
1.25	80	2.7/6	10.4	32	16.6	45.8				7.3/13	19.3	7.4	26.7		5.26	165
1.31	53	1.2/2	10.1	2	49.2	12.6			77	1.5/2	38.1	11.2	49.3		4.97	158
1.32	33		10.4	4	51.8	40.7			87	1.8/4	42.0	11.3	53.2		5.01	159
1.33	27	1.4/2	9.9	9	39.5	70.0	tmOut	tmOut	83	2.4/6	41.8	11.4	53.2	1	4.94	158
1.34	10	1	14.8	16	42.4	165			97	3.4/9	43.3	11.7	54.9		5.33	167
1.35	17		15.6	32	142	793			87	3.4/9	132	20.0	152		5.9	173

Table 7: The results of Experiment 1.2 and 1.3 for pruned ontologies

	GA-pruned			**SMT-pruned**					**Hybrid-pruned**					**FD-pruned**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
1.21	100	1.8/2	4.8	2	11.9	4.9	533.1	549.9	100	2.0/2	6.6		10.9		0.24	73
1.22	97	2.7/4	4.7	4	13.1	3.2	367.7	384.0	97	3.1/4	7.1	4.3	11.4		0.23	73
1.23	100	3.2/6	5.2	8	11.7	12.2	837.5	861.5		5.0/8	8.5		12.8	1	0.39	72
1.24	97	3.6/7	7.5	16	11.9	13.2	tmOut	tmOut	100	7.9/12	10.1	4.6	14.7		0.53	79
1.25		2.9/6	5.3	32	14.3	20.0				9.7/19	12.2	4.8	17.1		0.84	85
1.31	63	1.4/2	6.3	2	22.0	6.0	1237	1265	100	1.7/2	17.5	5.8	23.3		0.39	73
1.32	40	1.3/3	5.7	4	25.1	19.7			97	2.6/4	21.3	6.1	27.4		0.53	79
1.33	27	1.1/2	7.4	9	25.2	25.7	tmOut	tmOut	100	3.8/7	24.9	7.0	31.9	1	1.12	90
1.34	13	1	8.0	16	24.7	74.2				4.7/10	28.9	7.5	36.4		1.72	104
1.35			9.8	32	77.9	220			97	5.1/13	88.4	12.6	101		1.72	102

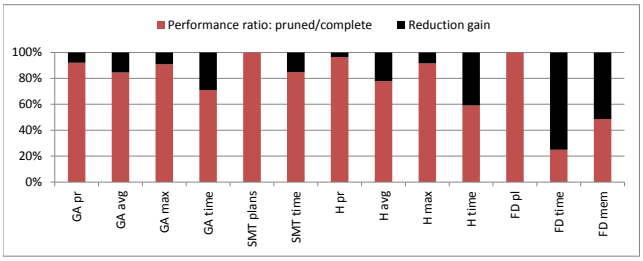


Figure 7: The influence of GDB reduction on the planning efficiency in Experiment 1. On the X-axis, from left to right: GA probability, average and maximal number of plans, and runtime; SMT plans and time; Hybrid probability, average and maximal number of plans, and time; FastDownward plans, time, and memory.

ceptable low (from 1.5 to 1.9 sec) and is fully compensated by the delivered benefits in the planners behaviour.

The two next experiments concern instances where the number of extended branches is scaled from 1 to 5. These yield from 2 to 32 plans of length 10 (Exp. 1.2) and of length 12 (Exp. 1.3). The benchmark parameters and features are given in Table 5, while the results for complete and reduced ontologies are given in Tables 6 and 7, respectively. In comparison to the benchmarks of Exp. 1.1, these of Exp. 1.2 and 1.3 are harder to solve due to the increased length of the plans.

It is also easy to see that the benchmarks of group 1.3 are more difficult than these of group 1.2, which results in higher computation times as well as lower probabilities and finally in less plans found by the GA and Hybrid planners. Surprisingly, the FD tool seems to be invulnerable to the length of the plans, because its runtime and memory consumption is not higher than in Exp. 1.1. Overall, the general trends of the planners' behaviour are similar to those of Exp. 1.1. After the ontology reduction the planning performance increases while the GDB pruning time stays low, between 1.4 and 1.7 sec.

Figure 7 summarizes the influence of the reduction on the planners' performance taking into account all benchmarks of Exp. 1. The black bars show the relative improvement of the planners behaviours taking into account all the assessed features. It is easy to observe that the reduction has the highest

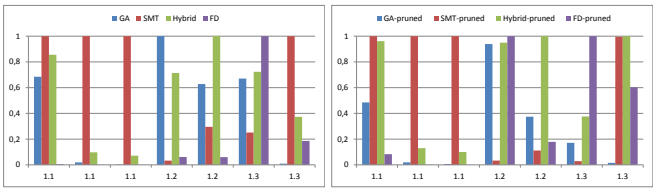


Figure 8: Experiment 1. Comparison of the planners efficiency using the score function.

impact on the FD's consumption of time and memory,but also on runtime of GA and Hybrid. One can observe also that all features, but the number of plans found by SMT and FD, are slightly improved.

In general, our goal is to find as many plans as possible. Thus, to compare the overall planner's efficiency taking into account also the number of plans found, we use the score function that rewards the results with many plans found. For each method (x) and each benchmark group (i) we calculate the value of the score function M^i_x which inputs (a sum of) the probabilities, the numbers of plans found, and the computation times. Then, the obtained values are normalized such that the value 1 is assigned to the results of the best method according to the value of M^i_x, computed as follows:

$$planReward^i_x = \begin{cases} 1 & \text{for } avgPlans^i_x \leq 0.5 \\ 2 & \text{for } 0.5 < avgPlans^i_x \leq 0.75 \\ 3 & \text{for } avgPlans^i_x > 0.75 \end{cases}$$

$$M^i_x = \frac{planReward^i_x * probability^i_x * maxPlans^i_x * avgPlans^i_x}{computationTime^i_x}$$

$$Eff^i_x = \frac{M^i_x}{max(M^i_{Hybrid}, M^i_{GA}, M^i_{SMT}, M^i_{FD})}$$

for $x \in \{Hybrid, GA, SMT\}$ and $i \in \{1, \ldots, numOfGroups\}$, where *avgPlans* and *maxPlans* are the average and the maximal number of plans found, respectively, and *numOfGroups* is the total number of benchmark groups from all experiments. The meaning of the remaining variables is consistent with their names.

The comparison of the planners efficiency using the score function is presented in Figure 8. It is easy to observe a great advantage of SMT in Experiment 1.1, since it is able to found all the plans, much more than any other planning method. Moreover, after the ontology reduction, Hybrid and FD are superior in the Experiments 1.2 and 1.3.

7.3.2. Experiment 2: scaling the lengths of the plans

In the following experiment we change values of parameters *len* and $n_{|EW|}$ affecting the *lengths of plans*. In fact, our generated benchmarks satisfy the following property: $k = n_{|EW|} * len$, that is, the length of the plan is a product of the two parameter values. The benchmark definitions are given in Table 8. In Experiment 2.1 we scale the value of *len* from 2 to 8 obtaining benchmarks yielding four plans of length from 6 to 24. Every generated query demands three objects to be produced ($n_{|EW|} = 3$), each of them by a sequence of *len* service types.

Experiment 2.2 scales the parameters in an opposite way. It consists of eight benchmarks where $len = 3$, but the number of objects to be produced ($n_{|EW|}$) changes from 2 to 9. Note that the benchmark 707 yields very long plans ($k = 27$), and the search space size reaches even 2^{162}. The sizes of search spaces for complete and pruned ontologies are given in the last two rows of Table 8

The planning results for complete and pruned ontologies are provided in Tables 9 and 10, respectively. The obvious conclusion is that the longer plan, the worse planners efficiency. This follows directly from the APP complexity which can be roughly estimated as $|\mathbb{S}|^k$. Thus, with the length of the plan also increases the computation time, but the probability decreases, and so the number of plans found. However, it is worth noticing a subtle difference in the planners behaviour dealing with benchmarks of different groups. Comparing the GA performance on benchmarks of Exp. 2.1 and 2.2 one can conclude that GA is more sensitive to the length of branches (*len*) than to the number of branches ($n_{|EW|}$). In the first case GA is unable to found any solution for plans of length 15 ($len = 5$), while in the second benchmark group GA finds a plan with an acceptable probability even up to length 21.

The conclusion is just opposite for the SMT-based planner. It deals a little better with the first benchmark group than the second. SMT is able to found all plans within the given timeout up to the depth 18 for the first benchmarks group, but only up to the depth 15 for the second group. This behaviour can be explained by two main factors. Firstly, the encoding of the expected worlds is more expensive in the presence of higher number objects, and thus the formula is then harder to solve. Secondly, the longer branch, the more dependencies between service types, the more logical inference can be used by an SMT-solver for efficient plan search by early-pruning the solution space. The latter fact can be seen even more clear in analysis of Experiment 3.1 and 3.2, and in particular in the results of benchmarks 215 and 206.

The hybrid planner shows here a behaviour similar to GA, because it deals better with the second benchmark group. However, due to support of the SMT-component the Hybrid algorithm is able to find more and longer plans than GA.

The FD performance in this experiment is quite impressive. The memory consumption stays quite low and in most cases the runtime is lower than for any other method. This experiment confirms the conclusion that FD is not so vulnerable to the length of the plan contrary to the other planners, which is a great advantage. However, the main FD drawback of finding at most one plan only remains unchanged. However, according to the score function, FD is superior for the medium and high values of the scaled parameters what is confirmed by Figure 9a.

The results for reduced ontologies are again significantly improved, for a low price of GDB computation - about 1.5 sec. Figure 9b summarizes the influence of the ontology reduction on the planners efficiency, taking into account all benchmarks of Experiment 2.

Table 8: Experiment 2.1 (left) and 2.2 (right) - benchmark parameters. Complete ontologies size: $|\mathbb{O}| = 102, |\mathbb{S}| = 64$.

	$n_{ext} = 2, n_{\|ext\|} = 2, n_{\|EW\|} = 3, \|CAP\| = 4$						
id	**2.11**	**2.12**	**2.13**	**2.14**	**2.15**	**2.16**	**2.17**
len	2	3	4	5	6	7	8
k	6	9	12	15	18	21	24
$\|\mathbb{S}_{pr}\|$	10	15	20	25	30	35	40
$\|\mathbb{O}_{pr}\|$	12	18	27	36	45	49	60
sp	2^{36}	2^{54}	2^{72}	2^{90}	2^{108}	2^{126}	2^{144}
sp_{pr}	2^{20}	2^{35}	2^{52}	2^{70}	2^{88}	2^{108}	2^{128}

	$n_{ext} = 2, n_{\|ext\|} = 2, len = 3, \|CAP\| = 4$							
id	**2.21**	**2.22**	**2.23**	**2.24**	**2.25**	**2.26**	**2.27**	**2.28**
$n_{\|EW\|}$	2	3	4	5	6	7	8	9
k	6	9	12	15	18	21	24	27
$\|\mathbb{S}_{pr}\|$	12	15	18	21	24	27	30	33
$\|\mathbb{O}_{pr}\|$	15	18	22	26	33	33	39	43
sp	2^{36}	2^{54}	2^{72}	2^{90}	2^{108}	2^{126}	2^{144}	2^{162}
sp_{pr}	2^{22}	2^{35}	2^{50}	2^{66}	2^{83}	2^{100}	2^{118}	2^{136}

Table 9: The results of Experiment 2.1 and 2.2 for complete ontologies.

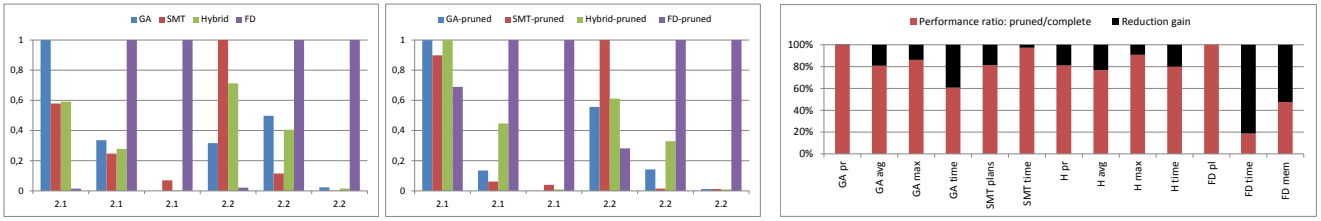
	GA			SMT					Hybrid					FD		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
2.11	100	3.5/4	3.9	4	4.3	0.7	2.7	7.7	100	3.3/4	2.8	3.9	6.7	1	14.9	320
2.12	83	1.8/4	6.4		18.9	5.8	63.8	88.5	97	2.5/4	12.1	6.9	19.1		9.32	255
2.13	53	1.1/2	12.1		42.0	29.0	954	1025	73	1.7/3	41.5	14.7	56.2		6.34	182
2.14	0	0	20.6		188	347	tmOut	tmOut	37	1.6/3	103.6	26.6	130.2		7.74	205
2.15			32.7		524	1468			13	1.3/2	429.9	44.6	474.5		9.08	218
2.16			50.7	1	1221	tmOut			7	1	414.1	61.4	475.5		7.29	195
2.17			73.9	0	tmOut				0	0	1225	95.6	1321		9.89	111
2.21	90	2.2/4	6.4	4	3.2	1.3	1.3	5.9	100	3.4/4	3.7	4.1	7.8	1	12.8	289
2.22	63	1.5/3	10.3		15.8	14.1	56.2	86.2	93	2.5/4	12.1	7.3	19.3		9.37	258
2.23	47	1.5/2	9.6		49.5	16.8	tmOut	tmOut	100	2.0/4	44.0	11.6	55.7		4.78	159
2.24	37	1.2/2	14.8		140	468			67	1.4/3	93.3	19.4	112.6		4.84	155
2.25	20	1	21.6	1	568	tmOut			50	1.5/3	225.0	27.7	252.6		6.05	179
2.26	27		45.0		750				30	1.3/2	493.2	40.9	534.1		5.44	165
2.27	3		41.8	0	tmOut				27	1.3/2	865	63	928		6.15	176
2.28			58.2						10	1	1317	91	1408		5.11	162

Table 10: The results of Experiment 2.1 and 2.2 for reduced ontologies.

	GA-pruned			SMT-pruned					Hybrid-pruned					FD-pruned		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
2.11	100	3.8/4	2.7	4	0.6	0.7	1.9	3.2	100	4	0.5	3.0	3.5	1	0.1	71
2.12	97	2.0/3	4.4		7.1	3.1	24.6	34.8		3.8/4	5.7	4.4	10.1		0.39	73
2.13	30	1.8/3	6.9		26.0	19.3	514.7	560		3.0/4	24.8	7.5	32.3		0.83	85
2.14	7	1	11.9		81.3	171	tmOut	tmOut	73	1.8/4	85.5	14.5	100		1.74	100
2.15	0	0	19.8		217	539			43	1.6/3	270	27.0	297		2.91	122
2.16			30.2	3	473	1422			27	1	510	42.0	552		3.21	127
2.17			49.3	1	1174	tmOut			30		1547	82.0	1629		5.3	162
2.21	100	3.1/4	3.2	4	0.7	0.7	0.9	2.2	100	4	1.2	3.5	4.7	1	0.25	73
2.22	90	1.9/4	5.1		6.8	3.6	22.1	32.4		3.5/4	5.9	4.5	10.4		0.39	72
2.23	50	1.5/3	5.7		20.6	16.1	tmOut	tmOut	93	2.9/4	22.7	6.0	28.7		0.55	79
2.24	23	1.4/2	9.5		53.6	44.9				2.3/4	43.6	10.3	53.9		0.7	81
2.25	17	1	11.8		174	251			73	1.5/3	126.6	15.1	141.7		1.3	92
2.26	10	1.3/2	15.6	3	428	1170			37	1.2/2	228.1	20.6	248.7		1.12	93
2.27	3	2	36.2	1	723.4	tmOut			13	1	463.9	34.0	497.9		1.52	100
2.28	0	0	35.5	0	tmOut				10		978	55.0	1033		1.99	109

7.3.3. Experiment 3: constant length and number of plans

Now we proceed to testing some combinations of parameters, which can be independently changed while maintaining the constant length of plans and the number of plans. Interest-



(a) Comparison of the planners efficiency using the score function.

(b) The influence of GDB reduction on the planning efficiency.

Figure 9: Graphical summary of Experiment 2.

ingly, the performance of planners changes as a result of these settings.

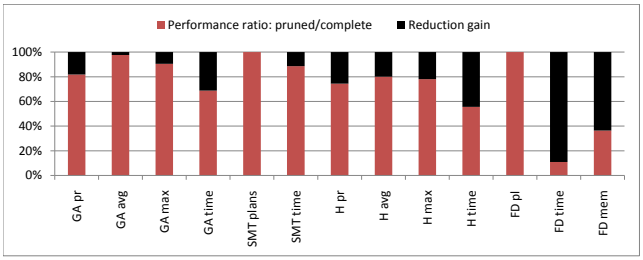


Figure 10: The influence of the GDB reduction on the planning efficiency in Experiment 3.

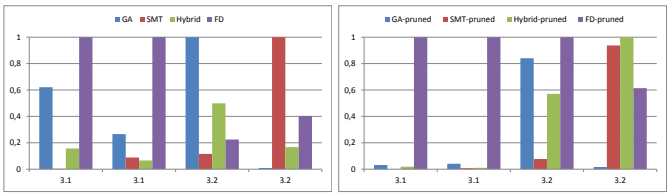


Figure 11: Experiment 3. Comparison of the planners efficiency using the score function.

Let us recall that $k = n_{|EW|} * len$, i.e., the length of a plan depends on the number of objects to be produced multiplied by the length of a sequence of service types delivering an object. In the first two experiments (3.1 and 3.2) we balanced the values of these two parameters in order to obtain plans of length 12.

The benchmark definitions and properties are given in Table 11. The results for complete and pruned ontologies are provided in Table 12 and Table 13, respectively. The GDB reduction time varies between 1.4 and 1.8 sec., similarly to the previous experiments.

The results of Experiment 3.1 and 3.2 provide a very interesting conclusion. One can observe that the subsequent benchmarks, where *len* is increased and $n_{|EW|}$ is decreased, are getting harder to solve by GA and Hybrid. This is confirmed by the decreasing probability. On the other hand, the same benchmarks are getting easier to solve by the SMT-based planner, as confirmed by the decreasing runtime. Thus, the conclusion is that SMT performs better for benchmarks composed of a smaller number of longer branches, while GA and Hybrid perform better for shorter branches even if the number of produced objects is high. This quite interesting behaviour can be explained twofold, like already mentioned in the discussion on Experiment 2. For SMT, a large number of expected objects makes the symbolic encoding more expensive and the resulted formula harder to solve. On the other hand, longer branches contain more dependencies that can be used to early pruning the solution space and boost the logical inference, what in consequence improve the symbolic planning.

The FD tool, similarly to GA and Hybrid, also performs better for shorter branches what is confirmed by relatively low runtimes and memory consumption for the first benchmark group increasing with the length of the branches for the subsequent instances. The GDB reduction improves the performance of all planning methods, but relatively the FD tool gains the most on the ontology pruning.

7.3.4. *Experiment 4: scaling the number of attributes, service types, and object types.*

Finally, let us discuss the parameters influencing neither the plan length nor the number of plans. We start with scaling the number of service types (Experiment 4.1) and object types (Experiment 4.2). The benchmark definitions are given in Table 14. The results for complete and pruned ontologies are provided in Table 15 and Table 16, respectively.

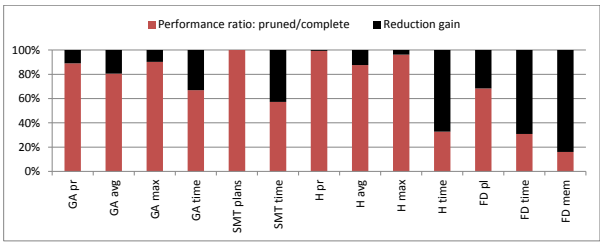


Figure 12: The influence of GDB reduction on the planning efficiency in Experiment 4.

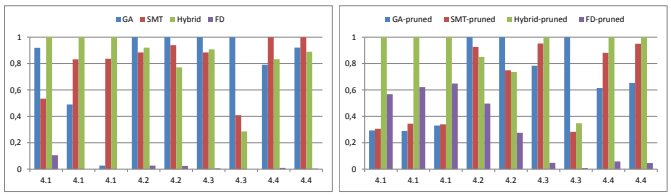


Figure 13: Experiment 4. Comparison of the planners efficiency using the score function.

Increasing the number of service types in the ontology makes the planning harder for all the methods, with respect to running times. Moreover, concerning GA, the probability and the number of plans found also drops with the increase of the number of service types. As to the performance of the FD tool, one can observe that it is able to found a plan only for the smallest ontologies. Otherwise, it runs out of memory. Thus, Experiment 4.1 shows a great advantage of Planics for planning in large ontologies.

The number of object types in the ontology has much less influence than the number of service types on the planning, what is confirmed by the results of Experiment 4.2. The planning times grow slightly, but the probabilities and numbers of plans found remain comparable. This can be explained by the fact that a fixed number of service types involves only a constant number of object types. All the 'excess' object types introduced are not associated with any services, so the planners do not consider them.

The ontology pruning reduces the ontologies to the relevant 'core', so the results are similar irrespectively of the number of service and object types in the full ontology. Again, the overhead associated with the pruning is relatively small (between 1 and 2 sec.) in comparison with the efficiency gain.

Table 11: Experiments 3.1 and 3.2 - benchmark parameters for k=12. The complete ontologies size is $|\mathbb{O}| = 102$, and $|\mathbb{S}| = 64$.

	$n_{ext} = 0, n_{\|ext\|} = 0, \|CAP\| = 1$						$n_{ext} = 2, n_{\|ext\|} = 3, \|CAP\| = 9$				
id	**3.11**	**3.12**	**3.13**	**3.14**	**3.15**	**3.16**	**3.21**	**3.22**	**3.23**	**3.24**	**3.25**
len	1	2	3	4	6	12	1	2	3	4	6
$n_{\|EW\|}$	12	6	4	3	2	1	12	6	4	3	2
$\|\mathbb{S}_{pr}\|$	12	12	12	12	12	12	16	20	24	28	36
$\|\mathbb{O}_{pr}\|$	13	17	16	20	17	21	13	23	28	37	46

Table 12: The results of Experiment 3.1 and 3.2, for complete ontologies.

	GA			**SMT**					**Hybrid**					**FD**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
3.11	100		6.6		55.1		tmOut	tmOut	100		15.9	7.1	23.0		4.73	156
3.12	90		8.8		45.1				90		32.0	9.9	41.9		6.82	186
3.13	67	1	10.8	1	44.3	-	1553	1597.3	67	1	33.5	11.7	45.1	1	6.65	194
3.14	60		13.2		51.4		250.3	301.7	57		32.6	13.5	46.1		5.08	161
3.15	47		12.6		50.8		52.0	102.8	47		38.2	15.1	53.3		12.89	283
3.16	3		13.7		59.9		5.6	65.5	0	0	33.1	15.8	48.9		22.23	569
3.21	90	2.4/6	6.6		41.7	15.4			97	2.4/6	11.0	7.1	18.1		2.88	130
3.22	97	2.0/5	8.3		35.8	74.6	tmOut	tmOut		2.6/7	30.4	9.5	39.9		6.58	195
3.23	17	1	10.0	9	52.2	88.1			83	2.9/8	42.5	11.5	54.0	1	4.8	158
3.24	7		12.7		48.6	81.3			57	2.6/5	56.1	16.1	72.3		8.21	214
3.25	0	0	14.3		45.4	97.7	336.2	479.3	17	1	45.7	16.0	61.7		7.61	200

Table 13: The results of Experiments 3.1 and 3.2, for reduced ontologies.

	GA-pruned			**SMT-pruned**					**Hybrid-pruned**					**FD-pruned**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
3.11	97		3.5		5.7		tmOut	tmOut	93		0.1	4.7	4.8		0.1	41
3.12	83		5.7		25.6				80		0.1	6.3	6.3		0.24	72
3.13	90	1	6.0	1	26.7	-	603	629.7	77	1	20.4	6.3	26.7	1	0.24	73
3.14	63		7.1		23.1		95.4	118.5	73		17.8	6.9	24.7		0.38	72
3.15	73		7.5		37.8		25.7	63.6	67		27.3	8.0	35.3		0.48	78
3.16	100		8.7		29.9		2.0	31.9	97		23.7	10.0	33.7		0.38	73
3.21	83	2.1/5	4.3		7.0	10.7			83	1.6/4	0.1	4.7	4.8		0.42	73
3.22	80	2.0/6	5.1		31.7	19.3	tmOut	tmOut	97	2.6/7	17.5	5.0	22.6		0.68	80
3.23	27	1.3/2	9.7	9	28.2	47.0			100	3.1/8	23.6	6.8	30.4	1	1.13	90
3.24	10	1.3/2	13.0		39.8	38.8	1728.4	1807		4.2/8	35.9	9.8	45.7		2.18	109
3.25	0	0	10.3		32.5	56.5	151	240	90	3.1/8	33.4	12.2	45.6		3.41	129

In Experiment 4.3 the number of objects in input lists is scaled. One can observe that the values of this parameter significantly influence the performance of the planners, except GA which behaves quite stable in comparison with the other methods. The FD tool appears to be very vulnerable to increasing the number of input objects, because it is able to found a plan only for the easiest benchmark. Ontology pruning improves the planners' performance, especially in the case of FD. After the reduction FD finds a plan in all but one hardest case.

Experiment 4.4 aims at scaling the number of object attributes. It has hardly any influence on the effectiveness of Planıcs modules, but the FD tool seems to be vulnerable to this parameter values what results in the increase of the time and/or memory consumption.

The graphical summary of Experiment 4 is presented in Figure 12 and 13.

7.3.5. General Conclusions

Concerning the characteristics of each planning method, some general conclusions can be drawn. As one could expect, each method has advantages and disadvantages, and the methods are often complementary. In the case of GA, its greatest advantage is a relatively short running time, which depends on the benchmark size only to a limited degree. The SMT-based planner finds all plans while it is the only one able to determine whether more plans exist or not. This is confirmed, e.g., by Experiment 1. The hybrid algorithm often benefits from both its components compensating their disadvantages, as shown in Experiment 4.

In general, all the Planıcs planners are very sensitive to the lengths of plans, contrary to the FD tool, as shows Experiment 2. Another advantage of FD is a low runtime, in many cases unavailable for other methods. To its drawbacks one should include the inability of finding more than one plan and poor results in dealing with complex services demanding many input objects, as confirmed by Experiment 4.4. The large number of services in the ontology is also prohibitive for FD and results in running out of memory quickly.

However, in many cases, the ontology pruning is to able to resolve these issues to a great extent and increase the planners performance. In Figure 14 we show the summary of the total reduction gain in all experiments. Surprisingly, the FD tool seems

Table 14: Experiments 4.1 (left) and 4.2 (right) - benchmark parameters

	$n_{ext}=2, n_{\|ext\|}=2,$ $n_{\|EW\|}=3, n_{\|\mathbb{O}\|}=102$		
id	**4.11**	**4.12**	**4.13**
$\|\mathbb{S}\|$	64	128	256
$\|\mathbb{S}_{pr}\|$	15	15	15
$\|\mathbb{O}_{pr}\|$	18	18	18

	$n_{ext}=2, n_{\|ext\|}=2, n_{\|EW\|}=3$ $\|CAP\|=4, \|\mathbb{S}\|=64$					
id	**4.21**	**4.22**	**4.23**	**4.24**	**4.25**	**4.26**
$\|\mathbb{O}\|$	102	152	202	252	302	352
$\|\mathbb{S}_{pr}\|$	10	10	10	10	10	10
$\|\mathbb{O}_{pr}\|$	12	12	13	15	14	16

Table 15: The result of Experiment 4.1 and 4.2, for complete ontologies.

	GA			SMT					Hybrid					FD		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
4.11	83	1.8/4	6.4	4	18.9	5.8	63.8	88.5	97	2.5/4	12.1	6.9	19.1	1	9.32	255
4.12	57	1.5/3	8.3		23.5	10.4	57.6	91.5		2.6/4	21.6	10.4	32.0	0	213	memOut
4.13	70	1.1/2	14.9		38.1	13.3	98.6	150	100	2.4/4	31.5	18.8	50.2		133	
4.21	100	2.9/4	4.2	4	4.0	1.3	3.8	9.0	100	3.8/4	3.2	3.9	7.1	1	15.1	322
4.22	97	2.7/4	4.3		4.2	1.4	2.4	8.0		3.5/4	3.3	4.1	7.3		4.49	150
4.23	100	3.5/4	5.2		4.7	0.9	4.0	9.6		3.7/4	3.8	5.3	9.2		4.77	119
4.24		3.2/4	6.1		4.3	0.4	2.9	7.7		3.6/4	4.0	6.3	10.3		6.1	115
4.25		2.8/4	6.7		5.2	2.1	5.2	12.5			3.5	7.1	10.6		7.93	120
4.26		3.4/4	6.8		6.1	1.3	3.1	10.4			4.1	7.3	11.4		10.6	117

Table 16: The result of Experiment 4.1 and 4.2, for reduced ontologies.

	GA-pruned			SMT-pruned					Hybrid-pruned					FD-pruned		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
4.11	97	2.0/3	4.4	4	7.1	3.1	24.6	34.8	100	3.8/4	5.7	4.4	10.1	1	0.39	73
4.12	80	1.8/4	4.7		6.4	5.7	20.8	32.9		3.6/4		4.5	10.2		0.38	
4.13	87	1.9/4	4.8		6.9	3.0	23.9	33.9		3.3/4	5.1	4.5	9.5		0.37	72
4.21	100	3.8/4	3.0	4	0.7	0.3	1.8	2.8	100	4	0.5	3.1	3.6	1	0.1	72
4.22		3.6/4	2.8		1.0	1.0	1.3	3.3		3.8/4	0.6	2.7	3.3		0.24	74
4.23		3.9/4	2.6		0.9	1.6	1.0	3.6		3.9/4					0.1	71
4.24		3.8/4	2.9		1.9	0.4	1.3	3.7			0.9	3.2	4.1		0.24	72
4.25		3.5/4	3.2		2.2	0.6	2.2	5.0			1.0	3.3	4.3			
4.26		4.0/4	2.9		1.7	1.0	1.5	4.2		3.8/4		3.1	4.1			

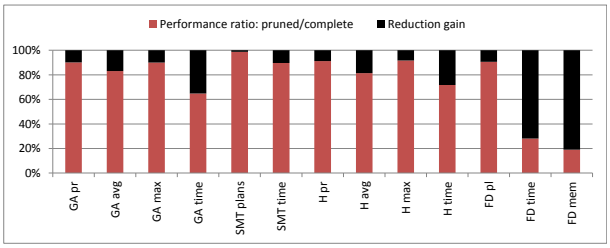


Figure 14: The total GDB reduction gain, in all experiments.

to benefit the most on the ontology pruning. Thus, the reduction not only improves performance of our planners, but could do it for external algorithms as shown at the example of FD. Sometimes, the reduction even enables the use of them, like in Experiments 4.1 and 4.3. For GDB the running times are relatively small and similar for all the benchmarks, only slightly depending on their size and other properties.

7.4. Comparison with other approaches

We have attempted to compare the performance of Planics with other tools. As the abstract planning is our original idea, there seem to be no other tools implementing it directly. In [6] there is a description of 7 SAT-based composition experiments performed with 413 *concrete* Web services, for which a SAT solver consumes between 40 and 60 sec. for every composition. However, only basic type matching has been considered with plans composed of only a couple of services, and the main objective was to find the shortest sequence satisfying the user query. We reproduced these experiments by modelling them in Planics, as follows. The web services became the service types, and the hierarchy of types used as web service parameters became the Planics object types. Our planner significantly outperforms those reported in the paper, finding a plan in only a few seconds out of which only a fraction of a second was the SAT time. In general, the presented solution is a simplified form of the Planics planning. Moreover, we search not only for a shortest solution, but we are aiming at finding all the plans.

In order to perform the second comparison, we translated Planics ontologies and queries to the PDDL language, and ap-

Table 17: Experiments 4.3 and 4.4 - benchmark parameters

	$\|\mathbb{S}\| = 64, \|\mathbb{O}\| = 102, n_{ext} = 2, n_{\|ext\|} = 2$					
id	**4.31**	**4.32**	**4.33**	**4.34**	**4.35**	**4.36**
n^{min}_{OinS}	1	3	4	5	6	10
$\|\mathbb{S}_{pr}\|$	14	14	14	14	14	14
$\|\mathbb{O}_{pr}\|$	15	30	39	48	48	90

	$\|\mathbb{S}\| = 64, \|\mathbb{O}\| = 102, n_{ext} = 2$				
id	**4.41**	**4.42**	**4.43**	**4.44**	**4.45**
n^{min}_{QA}	2	5	10	15	20
$\|\mathbb{S}_{pr}\|$	14	14	14	14	14
$\|\mathbb{O}_{pr}\|$	15	16	15	15	16

Table 18: The result of Experiment 4.3 and 4.4, for complete ontologies.

	GA			**SMT**					**Hybrid**					**FD**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
4.31	100	4.8/9	3.8	9	4.3	1.5	6.9	12.7	100	7.2/9	3.7	3.9	7.6	1	4.89	160
4.32	90	2.9/7	7.4		15.6	3.2	13.4	32.2	93	4.5/9	9.5	4.8	14.3		116	
4.33		5.2/8	6.9		24.9	7.2	24.4	56.5		5.8/8	19.1	6.8	25.9		175	
4.34	100	4.7/7	8.1		54.4	12.3	23.0	89.8	100	5.4/7	37.6	8.3	46.0	0	115	memOut
4.35		4.0/7	8.1		64.6	14.1	26.7	105.4		5.5/8	35.9	8.5	44.4		115	
4.36	97	3.3/6	15.2		814	138	51.0	1003		3.5/7	482	23	505		127	
4.41		4.8/9	3.8		4.3	1.5	6.9	12.7		7.2/9	3.7	3.9	7.6		4.89	160
4.42		5.1/8	6.8		4.5	2.0	3.3	9.8		6.3/9	3.5	6.8	10.3		4.46	155
4.43	100	5.2/8	4.6	9	5.0	3.5	4.1	12.6	100	7.2/9	4.5	4.3	8.8	1	5.77	174
4.44		5.5/9	6.3		4.3	2.8	4.5	11.6		6.9/9	5.3	5.7	11.0		26.2	244
4.45		5.6/9	5.7		6.3	3.6	5.6	15.5		6.3/9	5.2	5.3	10.6		50.8	109

Table 19: The result of Experiment 4.3 and 4.4, for reduced ontologies.

	GA-pruned			**SMT-pruned**					**Hybrid-pruned**					**FD-pruned**		
id	**pr. [%]**	**plans avg/max**	**time [s]**	**plans**	**first [s]**	**other [s]**	**unsat [s]**	**Σ [s]**	**pr. [%]**	**plans avg/max**	**smt [s]**	**ga [s]**	**Σ [s]**	**plan**	**time [s]**	**mem [MB]**
4.31		6.0/9	3.4		0.8	1.9	2.5	5.3		7.9/9	0.8	3.0	3.8		0.39	73
4.32		4.2/9	4.2		6.0	1.6	4.2	11.8		5.2/9	4.1	3.7	7.8		1.28	94
4.33	100	6.0/9	5.6	9	23.5	4.9	9.0	37.3	100	6.0/9	7.4	5.3	12.7	1	2.93	126
4.34		5.7/9	9.5		26.0	5.3	8.1	39.4		5.9/9	12.1	6.6	18.7		7.76	203
4.35		5.6/9	5.8		41.5	8.3	13.3	63.2		5.7/9	9.2	6.6	15.8		6.3	191
4.36			11.9		513	125	48.0	686		5.1/8	122	17	139	0	310	memOut
4.41		6.0/9	3.4		0.8	1.9	2.5	5.3		7.9/9	0.8	3.0	3.8		0.39	73
4.42		6.5/9	5.4		1.6	2.9	3.4	7.9		7.3/9	1.4	5.0	6.4		0.4	77
4.43	100	5.7/8	3.4	9	1.1	1.9	2.5	5.5	100	7.9/9	1.1	3.5	4.7	1	0.39	76
4.44		6.5/9	4.9		2.3	3.4	1.9	7.6			2.1	4.6	6.7		21.55	186
4.45		5.7/9	4.6		1.9	1.9	4.3	8.1		7.8/9	1.5	4.6	6.1		0.39	76

plied the Fast Downward tool [18, 64] as a planner. FD is one of the best planning tools (available for free), following the results of International Planning Competition 2011. PDDL [65] is one of the leading standards in the AI planning community. A PDDL planning problem definition is divided into two parts. The first one is a domain description which consists of, among others, definitions of types, predicates, and actions. The second one is a problem description defining objects, initial conditions, and goal states. Thus, in order to fairly confront Planics and FD we implemented a tool called Planics2PDDL. It takes an ontology and a user query as input and translates them into PDDL domain and problem files, respectively. The translation schema is the following. The object types from Planics ontology, as well as the (abstract) values of their attributes are encoded using PDDL predicates. The services are mapped as PDDL actions, while the query is translated into PDDL initial and goal clauses. An example of the translation is given in part V of [19].

For the comparison we used the benchmarks generated by our Ontology Generator. The tests have been run on the same machine with 2000 sec. time-out and memory limit of 12GB. The results are presented and discussed in the former section. One has to admit, that if the goal is to find one plan as quickly as possible, the FD tool could be superior in many cases when it works on small or reduced ontologies. However, it is noteworthy that usually GA finds the first solution quite quickly, not later than after a third or a half of the total computation time, but, contrary to FD, it does not stop and looks for another solution.

We have also adapted the Low Level Petri Net Analyser tool (LoLA) to deal with the abstract planning problem, as reported in [66]. To this aim we implemented a translation of the Planics ontologies and queries to high level Petri net specification accepted by LoLA augmented with a reachability property to test. Our overall conclusions of dealing with LoLA are quite similar to those concerning FD. Both the tools can be sometimes more efficient than Planics modules when dealing with small ontolo-

gies and long paths, but they are unable to find more than one plan and to search for plans in large ontologies.

8. Final Remarks

We have presented three new solutions to the abstract planning problem, based on SMT, GA, and their hybrid combination. All the three methods have been combined with an ontology pruning method based on a graph database.

The key concept is that different abstract plans are found by dealing with equivalence classes of user query solutions represented by multisets of service types. In case of the SMT-based algorithm, our idea consists in introducing formulas encoding multisets which represent abstract plans, and block all the solutions corresponding to the plans already found.

The GA-based algorithm proved to be highly efficient thanks to generating only some of possible linearisations of multisets, and applying the fitness function and mutation operator developed solely to APP. In result, the similarity measure built in the fitness function enables finding several plans in a single run, and the algorithm prefers the individuals composed of service types which produce object types requested in a user query, directly or indirectly (via other service types).

Our extensive experimental study shows that the combination of GA- and SMT-based approaches produces fairly good results, in particular for instances hard to solve by each of these algorithms applied separately. These problems have usually large search spaces containing many solutions.

Moreover, a novel combination of our planning approaches with the ontology pruning method appears to be a very efficient method to abstract planning in Planics as well in the FD tool.

Concerning the future work, we plan to extend the graph database approach, so that the whole planning, including matching of pre- and post-conditions, would be performed by the database. We also plan to experimentally check how the algorithms perform in a typical setting in which the composition works, i.e., when the ontology is modified relatively rarely and to a minor degree, but there are many queries to be answered concurrently. We expect the graph-based approach to perform well in this case, because databases are optimised to work in such conditions.

References

[1] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

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

[3] J. Rao, X. Su, A survey of automated web service composition methods, in: Proc. of SWSWPC'04, Vol. 3387 of LNCS, Springer, 2005, pp. 43–54.

[4] G. De Giacomo, M. Mecella, F. Patrizi, Automated service composition based on behaviors: The roman model, in: A. Bouguettaya, Q. Z. Sheng, F. Daniel (Eds.), Web Services Foundations, Springer, 2014, pp. 189–214. doi:10.1007/978-1-4614-7518-7_8.

[5] Z. Li, L. O'Brien, J. Keung, X. Xu, Effort-oriented classification matrix of web service composition, in: Proc. of the Fifth International Conference on Internet and Web Applications and Services, 2010, pp. 357–362.

[6] W. Nam, H. Kil, D. Lee, Type-aware web service composition using boolean satisfiability solver, in: Proc. of the CEC'08 and EEE'08, 2008, pp. 331–334.

[7] D. Berardi, F. Cheikh, G. De Giacomo, F. Patrizi, Automatic service composition via simulation, Int. J. Found. Comput. Sci. 19 (2) (2008) 429–451. doi:10.1142/S0129054108005759.

[8] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, F. Patrizi, Automatic service composition and synthesis: the roman model, IEEE Data Eng. Bull. 31 (3) (2008) 18–22.

[9] D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, J. Skaruz, HarmonICS - a tool for composing medical services, in: ZEUS, 2012, pp. 25–33.

[10] D. Doliwa, et al., PlanICS - a web service compositon toolset, Fundam. Inform. 112(1) (2011) 47–71.

[11] A. Niewiadomski, W. Penczek, Towards SMT-based Abstract Planning in PlanICS Ontology, in: Proc. of KEOD 2013 International Conference on Knowledge Engineering and Ontology Development, 2013, pp. 123–131. doi:10.5220/0004514901230131.

[12] J. Skaruz, A. Niewiadomski, W. Penczek, Evolutionary algorithms for abstract planning, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), PPAM (1), Vol. 8384 of Lecture Notes in Computer Science, Springer, 2013, pp. 392–401.

[13] A. Niewiadomski, J. Skaruz, P. Switalski, W. Penczek, Concrete planning in planics framework by combining SMT with GEO and simulated annealing*, Fundam. Inform. 147 (2-3) (2016) 289–313. doi:10.3233/FI-2016-1409.
URL http://dx.doi.org/10.3233/FI-2016-1409

[14] L. M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proc. of TACAS'08, Vol. 4963 of LNCS, Springer-Verlag, 2008, pp. 337–340.

[15] D. Dasgupta, Z. Michalewicz, Evolutionary algorithms in engineering applications, Springer, 1997.

[16] J. Muszynski, S. Varrette, P. Bouvry, F. Seredynski, S. U. Khan, Convergence analysis of evolutionary algorithms in the presence of crash-faults and cheaters, Computers & Mathematics with Applications 64 (12) (2012) 3805–3819. doi:10.1016/j.camwa.2012.03.004.

[17] X. Li, Q. Zhao, Y. Dai, A semantic web service composition method based on an enhanced planning graph., ICEE, 2288-2291(2010) (2010). doi:10.1109/ICEE.2010.578.

[18] M. Helmert, The fast downward planning system, Journal of Artificial Intelligence Research 26 (2006) 191–246.

[19] Online supplementary materials, http://planics.uph.edu.pl/asoc/supplement.pdf (2016).

[20] J. Skaruz, A. Niewiadomski, W. Penczek, Automated abstract planning with use of genetic algorithms, in: GECCO (Companion), 2013, pp. 129–130.

[21] A. Niewiadomski, W. Penczek, J. Skaruz, Hybrid approach to abstract planning of web services, in: Service Computation 2015 : The Seventh International Conferences on Advanced Service Computing, 2015, pp. 35–40.

[22] S. Mitra, R. Kumar, S. Basu, Automated choreographer synthesis for web services composition using I/O automata, in: ICWS, 2007, pp. 364–371.

[23] V. Chifu, I. Salomie, E. St. Chifu, Fluent calculus-based web service composition - from OWL-S to fluent calculus, in: Proc. of the 4th Int. Conf. on Intelligent Computer Communication and Processing, 2008, pp. 161 –168.

[24] V. Gehlot, K. Edupuganti, Use of Colored Petri Nets to model, analyze, and evaluate service composition and orchestration, in: System Sciences, 2009. HICSS '09., 2009, pp. 1 –8. doi:10.1109/HICSS.2009.487.

[25] J. Rao, P. Küngas, M. Matskin, Composition of semantic web services using linear logic theorem proving, Inf. Syst. 31 (4) (2006) 340–360. doi:10.1016/j.is.2005.02.005.

[26] P. Traverso, M. Pistore, Automated composition of semantic web services into executable processes, in: The Semantic Web ISWC 2004, Vol. 3298 of LNCS, Springer, 2004, pp. 380–394.

[27] B. Schlingloff, A. Martens, K. Schmidt, Modeling and model checking web services, Electr. Notes Theor. Comput. Sci. 126 (2005) 3–26. doi:10.1016/j.entcs.2004.11.011.

[28] A. Lomuscio, H. Qu, M. Solanki, Towards verifying contract regulated service composition, Autonomous Agents and Multi-Agent Systems 24 (3) (2012) 345–373. doi:10.1007/s10458-010-9152-3.

[29] M. Elwakil, Z. Yang, L. Wang, Q. Chen, 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, Springer, 2010, pp. 182–194.

[30] L. Bentakouk, P. Poizat, F. Zaidi, Checking the behavioral conformance of web services with symbolic testing and an SMT solver, in: Tests and Proofs, Vol. 6706 of LNCS, Springer, 2011, pp. 33–50.

[31] M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, M. Rossi, SMT-based verification of LTL specification with integer constraints and its application to runtime checking of service substitutability, in: SEFM, 2010, pp. 244–254.

[32] G. Monakova, O. Kopp, F. Leymann, S. Moser, K. Schäfers, Verifying business rules using an SMT solver for BPEL processes, in: BPSC, 2009, pp. 81–94.

[33] F. Lecue, M. D. Penta, R. Esposito, M. Villani, Optimizing QoS-aware semantic web service composition., in: Proceedings of the 8th International Semantic Web Conference, 2009, pp. 375–391.

[34] I. Garibay, A. S. Wu, O. Garibay, Emergence of genomic self-similarity in location independent representations, Genetic Programming and Evolvable Machines 7(1) (2006) 55–80.

[35] S. V. Hashemian, F. Mavaddat, A graph-based framework for composition of stateless web services., in: ECOWS, IEEE Computer Society, 2006, pp. 75–86.
URL http://dblp.uni-trier.de/db/conf/ecows/ecows2006.html#HashemianM06

[36] A Graph-Based Web Service Composition Technique Using Ontological Information.

[37] C. B. Mahmoud, F. Bettahar, H. Abderrahim, H. Saidi, Towards a graph-based approach for web services composition, CoRR abs/1306.4280.
URL http://arxiv.org/abs/1306.4280

[38] H. Elmaghraoui, I. Zaoui, D. Chiadmi, L. Benhlima, Graph based e-government web service composition, CoRR abs/1111.6401.
URL http://arxiv.org/abs/1111.6401

[39] S. Deng, B. Wu, J. Yin, Z. Wu, Efficient planning for top-k web service composition, Knowledge and Information Systems 36 (3) (2013) 579–605. doi:10.1007/s10115-012-0541-6.
URL http://dx.doi.org/10.1007/s10115-012-0541-6

[40] H. N. Talantikite, D. Aissani, N. Boudjlida, Semantic annotations for web services discovery and composition, Computer Standards & Interfaces 31 (6) (2009) 1108 – 1117. doi:http://dx.doi.org/10.1016/j.csi.2008.09.041.
URL http://www.sciencedirect.com/science/article/pii/S0920548908001591

[41] I. Robinson, J. Webber, E. Eifrem, Graph Databases, O'Reilly Media, Inc., 2013.

[42] R. Angles, C. Gutierrez, Survey of graph database models, ACM Comput. Surv. 40 (1) (2008) 1:1–1:39. doi:10.1145/1322432.1322433.
URL http://doi.acm.org/10.1145/1322432.1322433

[43] S. Shetty, S. P. R, A. K. Sinha, Article: A novel web service composition and web service discovery based on map reduce algorithm, IJCA Proceedings on International Conference on Information and Communication Technologies ICICT (4) (2014) 41–45, full text available.

[44] G. Kardas, A. Goknil, O. Dikenelli, N. Y. Topaloglu, Model driven development of semantic web enabled multi-agent systems, Int. J. Cooperative Inf. Syst. 18 (2) (2009) 261–308. doi:10.1142/S0218843009002014.
URL http://dx.doi.org/10.1142/S0218843009002014

[45] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, T. Kosar, On the use of a domain-specific modeling language in the development of multiagent systems, Eng. Appl. Artif. Intell. 28 (2014) 111–141. doi:10.1016/j.engappai.2013.11.012.
URL http://dx.doi.org/10.1016/j.engappai.2013.11.012

[46] S. Getir, M. Challenger, G. Kardas, The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems, Int. J. Cooperative Inf. Syst. 23 (3). doi:10.1142/S0218843014500051.
URL http://dx.doi.org/10.1142/S0218843014500051

[47] A. Lomuscio, W. Penczek, B. Wozna, Bounded model checking for knowledge and real time, Artif. Intell. 171 (16-17) (2007) 1011–1038. doi:10.1016/j.artint.2007.05.005.
URL http://dx.doi.org/10.1016/j.artint.2007.05.005

[48] Web Service Modelling Ontology D2v1.0, http://www.wsmo.org/2004/d2/v1.0/ (2004).

[49] M. Klusch, B. Fries, K. Sycara, Owls-mx: A hybrid semantic web service matchmaker for owl-s services, Web Semantics: Science, Services and Agents on the World Wide Web 7 (2) (2009) 121 – 133. doi:http://dx.doi.org/10.1016/j.websem.2008.10.001.
URL http://www.sciencedirect.com/science/article/pii/S1570826808000838

[50] OWL 2 web ontology language document overview, http://www.w3.org/TR/owl2-overwiew/ (2009).

[51] R. Lara, D. Roman, A. Polleres, D. Fensel, A Conceptual Comparison of WSMO and OWL-S., in: L. J. Zhang (Ed.), ECOWS, Vol. 3250 of Lecture Notes in Computer Science, Springer, 2004, pp. 254–269.
URL http://springerlink.metapress.com/openurl.asp?genre=article{\&}issn=0302-9743{\&}volume=3250{\&}spage=254

[52] M. Klusch, A. Gerber, Semantic web service composition planning with owls-xplan, in: Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, 2005, pp. 55–62.

[53] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL - the Planning Domain Definition Language - version 1.2, Tech. Rep. TR-98-003, Yale Center for Computational Vision and Control (1998).

[54] T. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. S. Nau, D. Wu, F. Yaman, SHOP2: an HTN planning system, CoRR abs/1106.4869.
URL http://arxiv.org/abs/1106.4869

[55] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, J. Artif. Int. Res. 14 (1) (2001) 253–302.
URL http://dl.acm.org/citation.cfm?id=1622394.1622404

[56] Resource Description Framework: Concepts and Abstract Syntax, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (2004).

[57] J. Dunkel, C. Kleiner, On managing services in service-oriented architectures, in: E. Ariwa, E. El-Qawasmeh (Eds.), Digital Enterprise and Information Systems - International Conference, DEIS 2011, London, UK, July 20 - 22, 2011. Proceedings, Vol. 194 of Communications in Computer and Information Science, Springer, 2011, pp. 410–424.
URL http://dx.doi.org/10.1007/978-3-642-22603-8_37

[58] Semantic Annotations for WSDL and XML Schema. W3C Recommendation., https://www.w3.org/TR/sawsdl/ (2007).

[59] L. Mikulski, A. Niewiadomski, M. Piatkowski, S. Smyczynski, On generation of context-abstract plans, in: Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers, 2014, pp. 376–388.

[60] S. Ranise, C. Tinelli, The SMT-LIB format: An initial proposal, in: In Proc. of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning (Miami Beach, USA), 2003.

[61] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB standard – version 2.0, in: Proc. of the 8^{th} Int. Workshop on Satisfiability Modulo Theories (SMT'10), 2010.

[62] D. R. Cook, The SMT-LIBv2 Language and Tools: A Tutorial, http://www.grammatech.com/resource/smt/ jSMTLIBTutorial.pdf (2012).

[63] A. Biere, A. Cimatti, E. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: In Proc. of the ACM/IEEE Design Automation Conference (DAC), 1999, pp. 317–320.

[64] M. Helmert, Concise finite-domain representations for PDDL planning tasks, Artif. Intell. 173 (5-6) (2009) 503–535. doi:10.1016/j.artint.2008.10.013.

[65] A. E. Gerevini, P. Haslum, D. Long, A. Saetti, Y. Dimopoulos, Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners, Artificial Intelligence 173 (5) (2009) 619–668.

[66] A. Niewiadomski, K. Wolf, Lola as abstract planning engine of planics, in: Proceedings of the International Workshop on Petri Nets and Software Engineering, co-located with 35th International Conference on Application and Theory of Petri Nets and Concurrency (PetriNets 2014) and 14th International Conference on Application of Concurrency to System Design (ACSD 2014), Tunis, Tunisia, June 23-24, 2014., 2014, pp. 349–350.
URL http://ceur-ws.org/Vol-1160/paper26.pdf