Solving the abstract planning problem using genetic algorithms

Jarosław Skaruz[1], Artur Niewiadomski[1], Wojciech Penczek[1,2]

[1] Institute of Computer Science, University of Natural Sciences and Humanities, 3 Maja 54, 08-110 Siedlce, Poland,
jaroslaw.skaruz@uph.edu.pl, artur.niewiadomski@uph.edu.pl
[2] Institute of Computer Science, Polish Academy of Sciences, Jana Kazimierza 5, 01-248, Warsaw, Poland,
penczek@ipipan.waw.pl

Abstract: The paper presents a new approach based on genetic algorithms to the abstract planning problem, which is the first stage of the web service composition problem. An abstract plan is defined as an equivalence class of sequences of service types that satisfy a user query. Intuitively, two sequences are equivalent if they are composed of the same service types, but not necessarily occurring in the same order. The objective of our genetic algorithm (GA) is to return representatives of abstract plans without generating all the equivalent sequences. The paper presents experimental results compared with the results obtained from SMT-solver, which show that GA finds solutions for very large sets of service types in a reasonable time.

Keywords: abstract planning, genetic algorithms, web service composition

1 Introduction

The number of web services available in the Internet has recently increased tremendously [29]. The users may want to achieve some goals taking advantage of these services, but they also demand more sophisticated functionality from computer systems. Frequently, a simple web service does not realize the user objective, so a composition of them need to be executed to this aim. The problem of finding such a composition is hard and well known as the Web Service Composition Problem (WSCP) [3,1,25]. There is a number of various approaches to solve WSCP [5,4,7,8], some of them we discuss in the next section. In this paper, we follow the approach of our system PlanICS [12,13,14], which has been inspired by [1,2]. The main assumption is that all the web services in the domain of interest as well as the objects which are processed by the services, can be strictly classified in a hierarchy of *classes*, organised in an *ontology*. Another key idea is to divide planning into several stages. The first phase of the planning process works with types (classes), while the second one - in the space of *concrete services* (instances of classes). The first stage produces an *abstract plan*, which becomes a *concrete plan* in the second phase. Such an approach enables to reduce dramatically the number of concrete services which are taken into account. This paper focuses on the abstract planning problem only.

The current approaches to the abstract planning (see Section 2) behave nicely for small and medium size ontologies. However, for ontologies containing a large number of service types, the computational time could be very long or even prohibitive. In this paper we propose a new approach based on an application of GA. Despite of the fact that GA have been widely used to solving the concrete planning problem its application to the abstract planning is much more sophisticated. The main challenge in this work is related to the fact that the search space could contain only a few or just one feasible solution, which need to be found by the algorithm. On the other hand, in the concrete planning problem, initial population of individuals in GA represent only feasible potential solutions, which is not the case here. The abstract planning approach based on GA is much more involved and this is our first contribution. Our second contribution consists in the new representation of the abstract plans (by multisets of service types), which allows for pruning the state space from all the sequences that correspond to the generated abstract plans so far.

An individual of GA represents a multiset of service types and all the operators of GA are performed on this multiset. This feature of GA constitutes a great improvement in comparison to the linear form of service types. The main advantage of this approach is that the algorithm does not need to bother about the correct order of the service types represented. It means that in comparison to a linear representation of the individual, the offspring created through genetic operators do not have to contain service types in the correct order. Next, a linear form of an abstract plan is created using the heuristic procedure before the fitness function evaluation has been done. An abstract plan is defined by a multiset of service types such that its linear form satisfies a user query. If GA finds a new abstract plan, then it is stored. All the individuals in the subsequent iterations are then ’punished’ by decreasing their fitness value if they are similar to the abstract plan found. The individual fitness value is lowered proportionally to the similarity to the abstract plans stored. To the best of our knowledge, the above approach is novel, and as our experiments show is also very promising.

The rest of the paper is organized as follows. Related work is discussed in the next section. Section 3 discusses the abstract planning problem. In Section 4 we present how GA is applied to obtain abstract plans. Section 5 presents the ontology generator we use to generate the service types for GA and discusses the experimental results of our algorithm. The last section summarizes the results and shows some possible future work.

2 Related Work

Some approaches to the abstract planning are shortly discussed below. However, to the best of our knowledge neither of the existing algorithms uses GA. Peer [24] illustrates a plan-space based algorithm which improves the plan search with a feedback gained from a plan execution for the automatic Web Service Composition. A new framework for incorporating QoS in a dynamic workflow system is presented in [9]. This algorithm is actually a depth-first traversal of all service types with an intermediate pruning. The selection of the best workflow is done by evaluating the QoS constraints of each candidate. In [17] a dynamic service composition framework with two layers is presented. The semantics of the components and the user query is modeled and then in the second layer an execution path is discovered based on the query and the semantics of the components. In [27], the authors present a logic based planner for DAML-S services, which is the predecessor of OWL-S. The authors of [15,16] define a framework, called Dynamic Composition of Service (DynamiCoS) that aims at supporting the service composition on demand at a runtime.

The problem of the concrete planning has been recently also extensively studied in the literature. Besides various metaheuristics [10], there are many papers dealing with an application of non-deterministic algorithms, namely evolutionary algorithms. In [26] a simple GA was used to obtain a good quality concrete plan. The problem is also tackled with a multiobjective optimization genetic algorithm to find a set of optimal Pareto solutions from which the user can choose the most interesting tradeoff [11]. In [6] the authors applied GA to the concrete planning problem based on a delivered abstract plan. The number of genes is the same as the number of the abstract services in the abstract plan and each gene corresponds to an offer of a given abstract service from the abstract plan. In the experimental study the authors used 25 abstract services and up to 25 offers for each service. Their approach allows to find the optimum in 500 iterations of GA.

In [21] the authors used a genetic algorithm to one phase planning, which combines an abstract and a concrete planning. They studied a QoS-aware semantic web service composition and showed how to effectively compute optimal compositions of QoS-aware web services by considering their semantic links. In the fitness function they maximize semantic quality attributes, while minimizing the QoS attributes. The experiments were conducted using 500 offers for each of 500 abstract services.

The paper [23] presents an application of a combination of two algorithms, namely Tabu Search (TS) and GA. In this approach the idea consists in incorporating TS as a local procedure of GA in order to escape from a local minimum of GA. Their experimental results were compared to the results obtained from a standalone TS and GA. They show that a hybrid algorithm outperforms the two other approaches.

A combination of two different algorithms was also defined in [19]. The authors transformed the problem of a concrete planning into a selection of the optimal path in the weighted directed acyclic graph. Unfortunately, they used only 10 abstract services and 35 offers belonging to each of them. The proposed algorithm works better than a simple GA.

The idea of a multiset representation of an individual is not entirely new as it was presented in [18]. The authors have shown a new Proportional GA with a representation based on protein concetrations rather than on their ordering. As a result there was no fitness preasure for any special order of genes. Moreover, such an individual representation allows for an evolution until genes are distributed evenly in individuals. In [28] the authors model a problem of non-coding DNA in biological systems as a new floating representation in which built blocks of a problem are not fixed at a position within an individual. The experimental studies have shown that this representation allows to find better results than using standard representation with genes at fixed positions. Constrained optimization problems were also studied in the literature [20]. In many classical optimization problems a penalty function approach is used. However, the most difficult aspect of this approach is to find appropriate penalty parameters needed to guide the search towards the constrained optimum. The authors proposed a special form of a penalty function rather than a standard one. In their approach a penalty function is penalty parameter free.

3 Abstract Planning Problem

This section introduces the Abstract Planning Phase (APP) as the first stage of WSCP in the PlanICS framework. APP makes intensive use of the *service types* and the *object types* defined in the *ontology*. A service type represents a set of web services with similar capabilities, while the object types are used to represent data processed by the service types. The set of all object types is denoted by $\mathcal{T}$. The *attributes* are components of the object types, while a single attribute consists of a name and a type. The ontology defines the inheritance relation, such that a subtype of some base object type retains all the attributes of a base type, and optionally introduces some new attributes. The *objects* are instances of the object types and are distinguishable by unique identifiers. The set of all the objects is denoted by $\mathcal{O}$. The values of the attributes of an object determine its state. A set of the objects in a certain state is called a *world*. One of the crucial concepts of PlanICS is a world transformation, described in details in the next subsection.

The *abstract values* are another important idea in APP. Since for APP there is no need to know the exact states of the objects, it is sufficient to know only whether an attribute does have some value or it does not. Thus, we introduce the special functions *isSet* and *isNull* defining abstract values of the attributes of the objects, to be used in the specifications of the user queries and the service types.

3.1 User queries and service types

The main aim of PlanICS is to find a composition of web services, which allows to achieve a user goal. The user requirements are specified in a form of a *user query*. A user query specification as well as a service type specification, consists of three sets of objects: *in*, *inout*, and *out*, and two Boolean formulas over the attributes of the objects from these sets, namely *preCondition* and *postCondition* (*pre* and *post*, for short). More precisely, *pre* is defined over the attributes of the objects from *in* and *inout*, while *post* can involve also the objects from *out*. Before APP the *pre* and *post* formulas are reduced to their DNF forms, and the values of the attributes are mapped to the abstract values only. That is, in the general case, a reduced formula is a disjunction of conjunctions of literals without negations, where each literal is of the form *isSet*(*o*.*a*) or *isNull*(*o*.*a*) with *o* denoting an object and *a* its attribute. Thus, during APP we deal with sets of objects and such reduced formulas only. The interpretation of a single disjunct of a reduced formula and of the respective set of objects leads to obtaining an *abstract world*, i.e., a set of objects which attributes have abstract values: **set** or **null**. We simply say that an attribute is set or is null. Thus, an interpretation of a whole reduced formula constitutes a set of abstract worlds. As this paper deals with APP only, in what follows we use the notions of *worlds* and *values* instead of *abstract worlds* and *abstract values*, resp., when it is clear from the context.

Now, we are in a position to define the service types and the user queries. A service type *s* is a pair of world sets (W^s_{pre}, W^s_{post}), called the *input* and the *output worlds*, respectively. That is, a service type is an interpretation of the respective service type specification, such that the input worlds are defined by *in*, *inout*, and *pre*, while the output worlds are determined by *in*, *inout*, *out*, and *post*. Moreover, let $\mathcal{S}$ denote a set of all the service types defined in the ontology. The definition of a user query is similar: a user query *q* is a pair of world sets (W^q_{init}, W^q_{exp}), called the *initial* and the *expected worlds*, respectively. That is, a user query is an interpretation of the respective user query specification, such that the initial worlds are defined by *in*, *inout*, and *pre*, while the expected worlds are determined by *in*, *inout*, *out*, and *post*. Fig. 1 presents an example of a service type specification. *BookSelling* is a simple service type, which does not take any "read-only" object as an input (its *in* set is empty), modifies an object of type *Book*, and produces an object of type *Invoice* with all the attributes set. Note that in the absence of alternatives in the *pre* and *post* formulas, *BookSeling* defines exactly one input and one output world.

```
BookSelling = {
  in = Ø,    inout = { (Book, b) },   out = { (Invoice, i) },
  pre = isSet(b.title) and isNull(b.owner),
  post = isSet(b.title) and isSet(b.owner) and
         isSet(i.price) and isSet(i.address) and isSet(i.name)  }
```

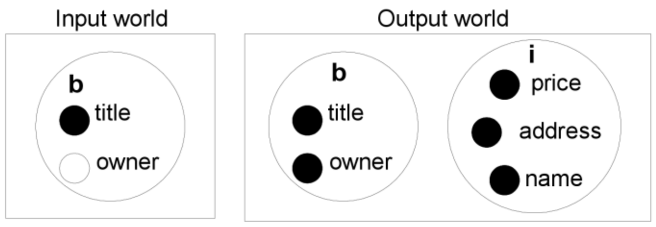


Figure 1: A simple service type specification and its interpretation as a pair of worlds

3.2 World transformations

A fundamental concept of PlanICS is a world transformation by a service of a given type. However, before we get to the details, we need to introduce a notion of the object states and the worlds compatibility. Assume we are given two objects o_1 and o_2 of some worlds, thus we know the (abstract) valuation of their attributes. We say that the state of the object o_1 *is compatible* with the state of the object o_2, if o_1 contains all the attributes of o_2 (thus both objects are of the same type, or o_1 is a subtype of o_2 type), and their valuations are not contradictory. This means that if o_1 is compatible to o_2, then every attribute of o_2 with the **set** (**null**) value, corresponds the same attribute of o_1, which is also set (null, resp.). Moreover, we say that a world w_1 is *compatible* to a world w_2, if both of them contain the same number of objects, and there exists a one-to-one mapping of the objects from w_1 to w_2, such that every object from w_2 corresponds to a compatible object from w_1. Finally, by a *sub-world* of a world *w* we mean a restriction of *w* to some subset of objects from *w*, and by the *size* of *w* we mean the number of the objects in *w*, denoted by |*w*|. Thus, a service of type *s* transforms a world *w* into *w*', denoted by $w \xrightarrow{s} w'$, if all of the following conditions hold:

- *w* contains a sub-world *IN* compatible with a sub-world of the input world of *s*, restricted to the objects from *in*,
- none of the objects of *IN* does change its state during the transformation,
- *w* contains a sub-world *IO* compatible with a sub-world of some input world of *s*, restricted to the objects from *inout*,
- *w*' contains a sub-world *IO* compatible with a sub-world of some output world of *s*, restricted to the objects from *inout*,
- *w*' contains a sub-world *OU* compatible with a sub-world of some output world of *s*, restricted to the objects from *out*,
- the sets of objects from *IN*, *IO*, *OU* are mutually disjoint, and *w* does not contain any of the objects from *OU*,
- $|w'|=|w|+|OU|$.

Intuitively, a service type *s* transforms *w* into *w*' by matching some sub-worlds of *w* (denoted by *IN* and *IO*) to its input world, “copying” all the objects from *w* to *w*', changing the states of the objects from *IO* according to the *post* formula, and creating new objects according to the set *out* and setting their states to be consistent with *post* (i.e., the sub-world *OU*). We refer to a world transformation by a service type *s* also as an *execution* of a service type *s* and by *transform*(*w*,*s*) denote the world *w*' if $w \xrightarrow{s} w'$.

Transformation sequences

Let *seq* = *(s₁, ..., s_k)* be a sequence of service types of length *k*, and let w_0 and w_k be worlds, for some $k \in \mathcal{N}$. We say that the sequence *seq* transforms the world w_0 into w_k, denoted by $w_0 \xrightarrow{seq} w_k$, if there exist worlds $w_1, ..., w_{k-1}$, such that $w_{i-1} \xrightarrow{s_i} w_i$, for every *i*=1,…,*k*.

A sequence *seq* of service types is called a *transformation sequence*, if there are worlds *w*,*w*', such that $w \xrightarrow{seq} w'$. The world *w*', i.e., the world obtained after the transformation of *w* by the transformation sequence *seq*, is called a *final world* of *seq*. The set of all the transformation sequences is denoted by $\mathcal{S}^\star$ while by M_{seq} we denote the multiset of the service types $[s_1, ..., s_k]$ of the transformation sequence *seq*. A transformation sequence *seq* that transforms a given world *w* is called a *transformation sequence for w*, and the process of transformation of *w* by *seq* is called the *execution of seq in w*.

Quasi-transformation sequences

Let *seq* = $(s_1, ..., s_k)$ be a sequence of service types of length *k*, and *w* be a world. We say that *seq* is a *quasi-transformation sequence for w*, if there exists $1 \leq j < k$ and $(s_1, ..., s_j)$ is a transformation sequence for *w*. Such a maximal *j* is called the *q-length* of the quasi-transformation sequence for *w*, and the prefix of *seq* of length *j* is called the *executable prefix of seq*. The final world of the quasi-transformation sequence for *w* of q-length is the world obtained by transformation of *w* by the executable prefix of *seq*. Intuitively, if *seq* is not a transformation sequence for *w*, but some non-empty prefix of *seq* is so, then *seq* is a quasi-transformation sequence for *w*.

Equivalent transformation sequences

Let *seq* = $(s_1, ..., s_k)$ and *seq'* = $(s'_1, ..., s'_k)$ be two transformation sequences of length *k*. Let us define a reflexive, transitive, and symmetric relation $\equiv \subseteq \mathcal{S}^\star \times \mathcal{S}^\star$ such that $(seq, seq') \in \equiv$, denoted by *seq*≡*seq*', if $M_{seq} = M_{seq'}$.

User query solutions

Let *seq* = $(s_1, ..., s_k)$ be a transformation sequence of length *k*, and q = (W^q_{init}, W^q_{exp}) be a user query. We say that the transformation sequence *seq* is a *solution of a user query q*, if there are worlds *w*,*w*', such that $w \xrightarrow{seq} w'$, $w \in W^q_{init}$, and $w' \in W^q_{exp}$. The set of all solutions of the user query *q* is denoted by QS_q.

Intuitively, a solution of the user query *q* is every transformation sequence that transforms some initial world into some expected world, defined by the user query *q*.

Abstract plans

Let $seq \in QS_q$ be a solution of some user query q. An abstract plan is a set of all solutions being equivalent to seq, i.e., it is the equivalence class $[seq]_{\equiv}$. An abstract plan $[seq]_{\equiv}$ is represented by the multiset of the service types M_{seq} for q.

Example 1 *Assume that Selling (S), Transport (T), Assembly (A) are service types, while Boards, Nails, and Doghouse are object types extending the object type Ware. The service type Selling is able to provide any Ware, Transport can deliver any Ware to the requested destination, and Assembly builds a doghouse using nails and boards. If the user wants to obtain a doghouse, there are several ways to reach this goal.*

The shortest solution is the sequence (S,T). This is the only solution of the abstract plan represented by the multiset [S,T]. Another possibility is (S,T,S,T,A), where the first pair (S,T) provides and transports boards while the second pair provides and delivers nails, which are finally assembled by A providing a doghouse. This solution constitutes another abstract plan represented by [A,S,S,T,T]. Note that there exists another equivalent solution, namely the sequence (S,S,T,T,A).

The next possible plan is represented by [A,S,S,T,T,T], when the requested doghouse is assembled elsewhere than at the client, and it has to be finally transported.

4 Application of GA to the abstract planning

The objective of GA is to find abstract plans for a user query q While GA maintains a population of individuals representing a multiset M of service types, it is essential to check whether M represents an abstract plan. To this aim, for M a sequence of service types seq_M is constructed according to the procedure *seqGen* (see Sec. 4.2). If seq_M is a solution to the user query q, then M represents a new abstract plan. In the next four subsections we describe in detail how GA works.

4.1 An abstract plan coding scheme

An individual is used for modelling an abstract plan we would like to find. A gene of an individual models a service type. So, the number of the genes of an individual is equal to the number of service types in an abstract plan. Let n denote the number of service types defined in the ontology, i.e., $n=|\mathcal{S}|$, and let $Num=\{0,1,\ldots,n-1\}$. We define a one-to-one function *stype*: $\mathcal{S} \longmapsto Num$, which to every service type assigns a natural number between 0 and $n-1$. Finally, in our implementation an individual is a multiset over *Num*.

All the individuals in the initial population of GA are generated randomly. This means that at the beginning of the algorithm the whole population contains multisets of service types, which do not necessarily represent abstract plans. One of the advantages of our approach is that while an individual is a multiset of service types, we do not need to care about the order of the service types within the individual. This non-standard form of a GA individual 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 the fitness function evaluation, a sequence of service types should be generated from an actual multiset. Since we do not generate all the sequences, the state space searched is dramatically reduced. This feature of GA allows us to obtain user query solutions in search spaces of sizes exceeding even 2^{100} (see Sec. 5).

4.2 Generating a sequence from a multiset

Although, an individual is a multiset of service types, at some points of our algorithm (like, for example, computation of a fitness value) we need to consider a transformation sequence built over the elements of the multiset. Obviously, we search for user query solutions, and therefore the sequences able to transform an initial world are of our particular interest.

```
Procedure seqGen($M$, $w_0$)
Input: multiset of service types: $M$, initial world: $w_0$
Result: (quasi) transformation sequence: $seq$, (q-)length of $seq$: $l$, final world of
        $seq$: $w$
begin
    $w \leftarrow w_0$;
    $seq \leftarrow \epsilon$ ; // empty sequence
    $l \leftarrow 0$;
    while $M$ contains $s$ that can be executed in $w$ do
        $seq \leftarrow seq \cdot s$ ; // append $s$ to $seq$
        $l \leftarrow l + 1$;
        $M \leftarrow M \setminus \{s\}$ ; // remove $s$ from $M$
        $w \leftarrow transform(w, s)$;
    end
    while $M$ is not empty do
        $M \leftarrow M \setminus \{s\}$ ; // remove some $s$ from $M$
        $seq \leftarrow seq \cdot s$ ; // append $s$ to $seq$
    end
    return $(seq, l, w)$
end
```

Algorithm 1. Proc. seqGen generating a sequence from multiset

The procedure *seqGen* allowing to obtain such sequences from a multiset is given in Alg. 1. In the successive iterations we build a resulting sequence by removing from the multiset a service type *s*, which is able to transform[1] a current world *w*, starting from some initial world w_0, randomly selected at the start of GA from W^q_{init}, of the user query *q*. Then, the current world becomes the one obtained from the transformation of *w* by *s*, and *s* is appended to the resulting sequence. If none of the service types remaining in the multiset can be executed in the current world, then they are copied in a random order at the end of the sequence. Besides the sequence *seq*, the procedure returns also a natural number *l*, and a world *w*, which

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

are used later to compute the fitness value of the individual. The world *w* is the final world of the sequence *seq*, while *l* is the (q-)length of *seq* if *seq* is a (quasi-) transformation sequence.

There are several reasons for the procedure not to consider all possible sequences that could be constructed from a given multiset. Firstly, for a given multiset of cardinality *k* the number of all possible sequences is equal to *k*!. Secondly, we prefer the sequences transforming an initial world of the user query. And finally, if the individual passes to the next generation, still it will be possible to construct another sequence from the same multiset.

4.3 Fitness function

To evaluate an individual, its fitness value should be calculated. The fitness function is defined in such a way that it leads to significant improvements of the initially randomly selected individuals, aiming at obtaining user query solutions.

Before we get to the details of the fitness function, we first need to define the notion of a *good service type*. Assume, we are given a sequence of service types of length *k* $seq = (s_1, ..., s_k)$, and a user query *q*. Let us consider service types s_i and s_j, where $i,j \in \{1,\ldots,k\}$, $i \neq j$, and $s_i \neq s_j$. By in_{s_i}, $inout_{s_i}$, out_{s_i}, $inout_q$, and out_q we denote the sets of objects used, modified, and produced by the services type s_i, and requested to be modified, and produced by the user query *q*, respectively. Moreover, let us define the function $\mathbf{T}:2^{O} \mapsto 2^{T}$, which with a set of objects assigns the set of the types of these objects. We say that s_i is a *good service type* for the sequence *seq* and the user query *q*, if $\mathbf{T}(inout_{s_i} \cup out_{s_i}) \cap \mathbf{T}(inout_q \cup out_q) \neq \emptyset$, or there exists s_j in *seq*, such that s_j is a good service type and $\mathbf{T}(inout_{s_i} \cup out_{s_i}) \cap \mathbf{T}(in_{s_j} \cup inout_{s_j}) \neq \emptyset$.

Intuitively, a service type s_i is good, if it produces the objects that can be a part of the expected world, or they can be an input for other good service types. The procedure *GST* computing a set of the good service types for a transformation sequence and a user query is given in Alg. 2.

Thus, an individual *M* is transformed to a sequence of service types seq_M, using the procedure *seqGen*, described in the previous subsection. Next, the fitness function, taking a triple (seq_M, l_M, w_M) returned by *sepGen* and an expected world[2] w_q as arguments, is calculated according to Eq. 1:

$$fitness_M = \frac{f_{w_M} * \delta + c_{w_M} * \alpha + l * \beta + g_{seq_M} * \gamma}{\delta * |w_q| + |w_q| * \alpha + k * \beta + k * \gamma}$$

where:

[2] Selected randomly from $W_{q,exp}$ at the start of GA

$f_{w_M}=|w_{sub}|$, where w_{sub} is a maximal sub-world of w_M compatible with a subworld of w_q,

c_{w_M} is the number of the objects from w_M, which types are consistent with types of the objects from w_q,

g_{seq_M} is the number of the good service types occuring in seq_M,

k is the length of seq_M, and

α,β,γ are parameters of the fitness function. In all the experiments presented in Sec. 5 we used the following values: $\alpha=0.7$, $\beta=0.1$, $\gamma=0.2$, and $\delta=0.1$.

After a user query solution has been found by GA and stored in the memory, the requirement for GA is to assure that other solutions remaining in the search space will be found. On the other hand, each individual that represents a solution equivalent to one of the already known should be eliminated. The latter is the task of the *measure of similarity* between the currently rated individual and the plans found so far. Obviously, the measure of similarity grows with the number of the service types common for the assessed multiset and one of the plans in the memory.

Let *Sol* denote a non-empty set of plans (in a form of multisets) found at some point of GA. Then, the measure of similarity of a multiset *M* is computed as follows:

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

Note that the similarity measure of a multiset identical to some plan is equal to 1, while 0 is the similarity measure of a multiset built over completely different types than these in the plan.

Finally, when there are solutions found, the fitness value of the individual *M* is calculated according to Eq. 3:

$$fitness_M^{Sol} = fitness_M * (1.0 - sim_M^{Sol})$$

The more the individual *M* is similar to some known plan, the more the value of sim^{Sol}_M decreases the fitness value of *M*.

```
Procedure GST($seq$, $q$)
Input: sequence of service types: $seq$, user query: $q$
Result: set of good service types occurring in $seq$: $GS$
begin
    $GS \leftarrow \emptyset$;
    $S \leftarrow \overline{seq}$ ; // set of all types occurring in $seq$
    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 $\mathcal{T}(inout_s \cup out_s) \cap \mathcal{T}(inout_q \cup out_q) \neq \emptyset$ then
            $GS \leftarrow GS \cup \{s\}$ ; // add $s$ to $GS$
        end
    end
    $S'' \leftarrow GS$;
    repeat
        $S' \leftarrow \emptyset$;
        $S \leftarrow \overline{seq} \setminus GS$;
        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 $\mathcal{T}(inout_s \cup out_s) \cap \mathcal{T}(in_g \cup inout_g) \neq \emptyset$ then
                    $S' \leftarrow S' \cup \{s\}$ ; // add $s$ to $S'$
                    break;
                end
            end
        end
        $GS \leftarrow GS \cup S'$ ; // add $S'$ to $GS$
        $S'' \leftarrow S'$;
    until $S' = \emptyset$;
    return $GS$
end
```

Algorithm 2. Proc. GST computing a set of good service types accurring in a sequence

4.4 Mutation operator

One of our contributions in this paper is a new mutation operator specialized to the problem considered, which takes advantage of the *good service type* concept. Therefore, a gene is mutated only if it does not represent a good service type, and if there exists a *good* service type for the considered sequence, generated by the algorithm *seqGen*. To this aim, one has to compute a set of all good service types for this sequence (see Alg. 3). If this set is not empty, then a randomly selected element of the set replaces the mutated gene. Thus, the mutation operator is not deterministic and it does not work in a greedy way.

```
Procedure mutGST($seq$, $q$)
Input: sequence of service types: $seq$, user query: $q$
Result: set of the good service types for $seq$ and $q$ to be used by the mutation
        operator: $GS$
begin
    $GS \leftarrow GST(seq, q)$;
    $S \leftarrow \mathbb{S} \setminus GS$;
    foreach $s \in S$ do
        if $\mathcal{T}(inout_s \cup out_s) \cap \mathcal{T}(inout_q \cup out_q) \neq \emptyset$ then
            $GS \leftarrow GS \cup \{s\}$ ; // add $s$ to $GS$
            continue;
        end
        foreach $g \in GS$ do
            if $\mathcal{T}(inout_s \cup out_s) \cap \mathcal{T}(in_g \cup inout_g) \neq \emptyset$ then
                $GS \leftarrow GS \cup \{s\}$ ; // add $s$ to $GS$
                break;
            end
        end
    end
    return $GS$
end
```

Algorithm 3. Proc. mutGST computes a set of the good service types for a sequence and a user query to be used by the mutation operator

5 Experimental Results

We have evaluated our algorithm using the ontologies, the user queries, and the abstract plans generated by our software - Ontology Generator (OG, for short). Each ontology contains an information about the services and the object types. OG generates the ontologies in a random manner such that semantic rules are met. Morever, OG provides us with a user query which corresponds to services and object types contained in the ontology. Each query is also generated randomly in such a way that the number of various abstract plans equals to the value of a special parameter of OG. This guarantees that we know a priori whether GA finds all solutions. The remaining parameters of the generator are: the number of various object types, the minimal and maximal number of the object attributes, the number of service types, the minimal and maximal number of objects in the sets *in*, *inout*, and *out* of the service types, the number of the objects required by a user, and the number of the services in an abstract plan. Thanks to many different settings of OG, one can receive such data, which are helpful for checking how well GA scales for finding optimal solutions. The scalability can be examined by fixing different sizes of services in the ontology and the number of services in the abstract plans.

The experimental study was divided into two stages. In the first one, we have tuned the values of all GA parameters.

The tuning procedure is as follows. We select a parameter, the other parameters are set to the typical values, and several experiments are conducted in order to find the best value of the selected parameter, which is then fixed and set to this value. This procedure is repeated in the same way for all the remaining parameters, where the values of the fixed parameters are not changed anymore. The number of the individuals is equal to 1000, probability of mutation and two-point crossover are 5% and 95%, respectively. The roulette selection operator was used in all experiments. Each benchmark has been stopped after 50 iterations. The experiments were run on a standard PC computer with two cores 2.8GHz CPU and 8GB RAM. The table 1. presents experimental setup.

Table 1: Experimental setup

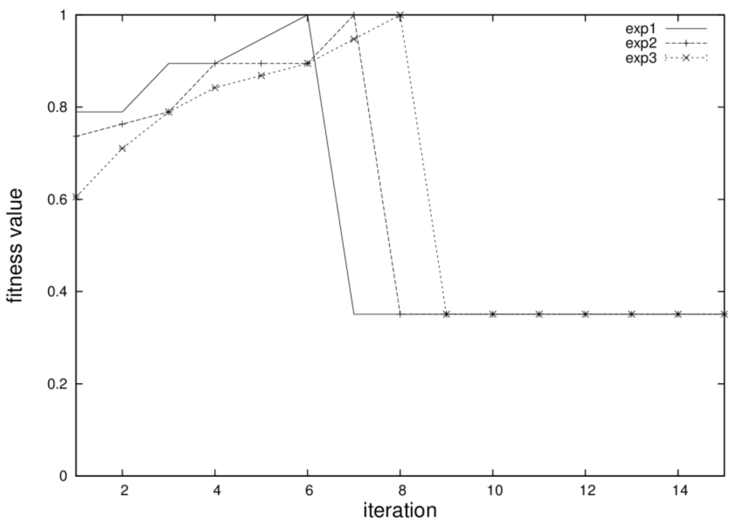
Exp. No.	Plan length	Number of solutions	Number of service types	Search space size
1	6	1	64	2^{36}
2			128	2^{42}
3			256	2^{48}
4		10	64	2^{36}
5			128	2^{42}
6			256	2^{48}
7	9	1	64	2^{54}
8			128	2^{63}
9			256	2^{72}
10		10	64	2^{54}
11			128	2^{63}
12			256	2^{72}
13	12	1	64	2^{72}
14			128	2^{84}
15			256	2^{96}
16	15	1	64	2^{90}
17			128	2^{105}
18			256	2^{120}

Table 2: Experimental results

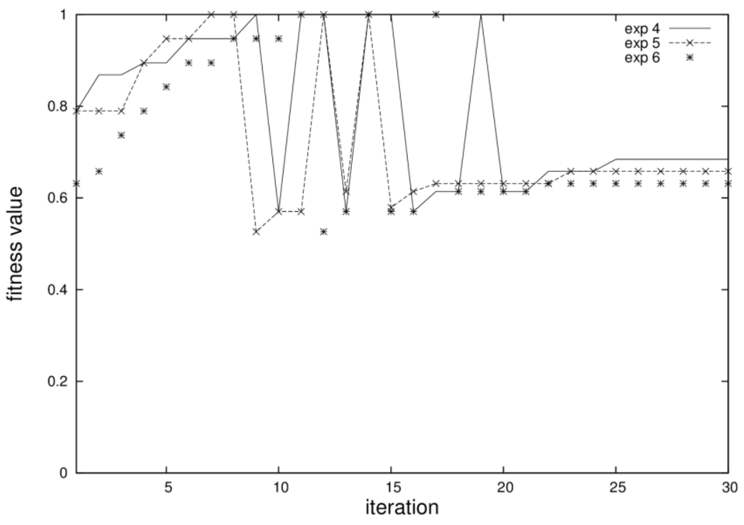
Exp. No.	Prob [%]	Max sol.	Avg sol.	GA first	GA next	GA [s]	SMT first[s]	SMT next [s]	SMT unsat [s]	SMT total[s]
1	100	1	1	6	—	6	4.28	—	2.9	8.09
2	100	1	1	6	—	7.5	7.76	—	5.87	14.83
3	100	1	1	8	—	10	11.19	—	7.57	20.57
4	100	6	4.1	9	11	6	4.89	4.99	4.95	12.86
5	100	6	3.7	7	8	7.5	5.95	6.0	9.63	20.5
6	100	4	2.6	11	13	11	13.85	14.68	17.56	38.47
7	100	1	1	10	—	11	21.05	—	25.57	47.63
8	100	1	1	10	—	15	39.48	—	41.75	83.04
9	100	1	1	12	—	22	94.55	—	77.65	174.5
10	80	2	1	15	17	12	17.84	19.44	161.1	239.2
11	60	2	0.8	18	23	15	34.19	44.76	276.1	341.8
12	50	1	0.5	26	—	22	65.85	67.87	542.5	669.7
13	100	1	1	15	—	16	91.9	—	429.8	523.2
14	100	1	1	21	—	22	126.4	—	1141	1270
15	90	1	0.9	29	—	34	213.1	—	4*>1800	
16	80	1	0.8	21	—	28	191.7	—		
17	20	1	0.2	22	—	35	425.7	—		
18	20	1	0.2	21	—	49	552.2	—		

Table 2 presents the summary of all the 18 experiments comparing the efficiency of our GA with an SMT-based planner [22]. The columns from left to right display: the experiment labels, the number of service types in the plans, the number of the existing plans, the total number of the service types, and the search space size. The next six columns contain the following GA results: the probability of finding a solution, the maximum and average number of the solutions found, the number of iterations needed to find the first and the second abstract plan, and the total GA runtime. The last four columns contain the times consumed by the SMT-based planner, in order to: find the first and the second solution, search the whole state space to ensure that there is no more plans, and the total SMT-planner runtime.

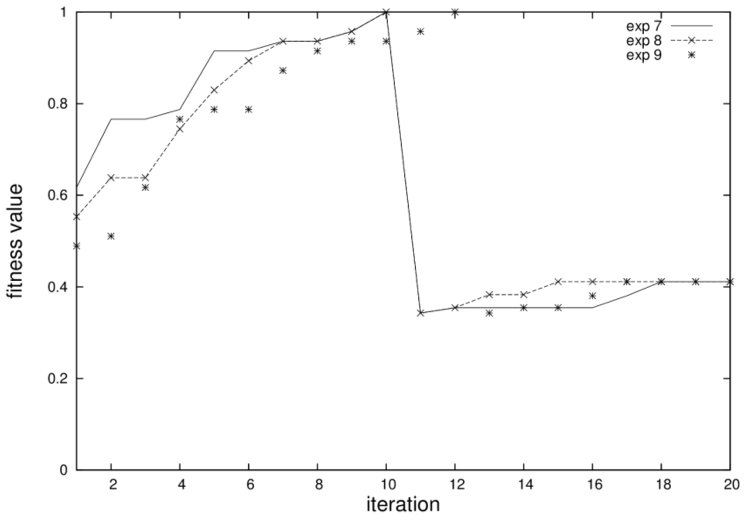
The experimental results can be summarized in the following way. As far as the time needed to find the first plan is concerned, the approach based on GA outperforms that based on SMT, because GA finds it dozen of times faster. However, the probability of finding a solution by GA decreases along with the increase of the length of abstract plans, similarly as for the average and the maximum number of the solutions found. Obviously, the more service types in an abstract plan the longer runtime of both the planners. In all the experiments the time required to find a solution by GA is below 21 seconds, while SMT needs even over 500 seconds. On the other hand, the SMT-based planner finds all the solutions in each run. Moreover, it is able to check that all possible abstract plans have been found.


exp1
exp2
exp3
fitness value
iteration
0
0.2
0.4
0.6
0.8
1
2
4
6
8
10
12
14

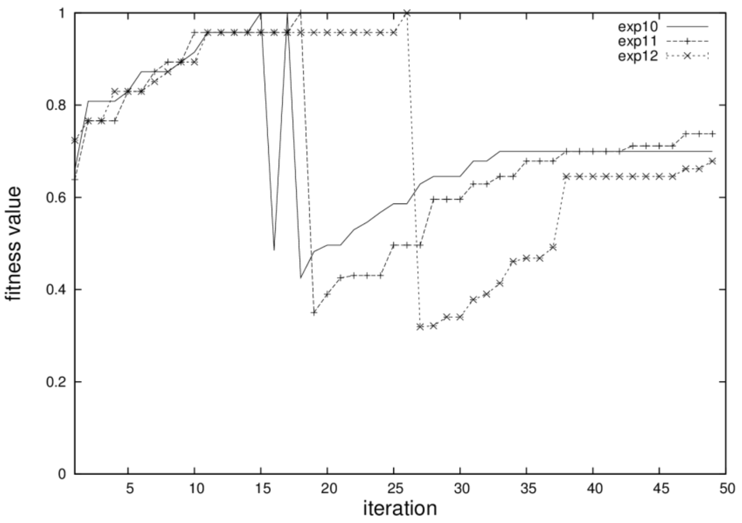
(a)


exp 4
exp 5
exp 6
fitness value
iteration
0
0.2
0.4
0.6
0.8
1
5
10
15
20
25
30

(b)



(c)



(d)

Figure 2: GA performance for 64, 128, and 256 service types. *exp*1−*exp*3 (a), *exp*4−*exp*6 (b), *exp*7−*exp*9 (c), *exp*10−*exp*12 (d)

Figure 2 presents the fitness value of the best individuals obtained in the experiments *exp1-exp12*. The most important observation resulting from the interpretation of the charts is a significant reduction of the best individual fitness value just after finding the solution, by the similarity measure. The fitness of the best individual chart shape can be viewed as a proof that our algorithm works as we have expected.

In the case of the first three experiments (Fig. 2a) the plans were found quite quickly and in all the runs of GA we have obtained solutions. Since the similarity measure works nicely, in the experiments *exp*4−*exp*6 (Fig. 2b) we obtained a number of solutions within the first 20 iterations. In the experiments *exp*7−*exp*9 (Fig. 2c) the fitness values of the initially generated individuals are in the range between 0.5 and 0.62. In the subsequent iterations GA finds better potential solutions. Finally, optimum is found in the 10th and the 12th iteration. In the experiments *exp*10−*exp*12 (Fig. 2d) GA obtains solutions before the 26th iteration. After one solution has been found the algorithm tries to find the next one, as the fitness value of the best individual increases in subsequent iterations. However, due to a much larger search space than in the experiments *exp*4−*exp*6, only in the experiment *exp*10 the next solution has been found.

6 Conclusions

In the paper we presented a novel approach to the abstract planning problem with use of a genetic algorithm. Optimal solutions representing abstract plans have been found in each instance of the problem. This was achieved thanks to the special forms of the fitness function and the mutation operator. To overcome the problem of generating similar abstract plans, we have used multisets of service types for representing abstract plans as well as individuals of GA. Such a representation allows to generate only one solution from the set of all the equivalent ones. The experimental results give a clear evidence that our approach is quite efficient and allows to find abstract plans containing as many as 15 service types. In comparison to the results obtained using an SMT-solver, GA finds solutions in a much shorter time, which makes it a suitable tool for deployment in information systems.

Acknowledgments

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

References

1. S. Ambroszkiewicz. Entish: An approach to service description and composition. *ISBN 83-910948-7-1, ICS PAS*, 2003.
2. S. Ambroszkiewicz. Entish: A language for describing data processing in open distributed systems. *Fundam. Inform.*, 60(1-4):41–66, 2004.
3. M. Bell. *Introduction to Service-Oriented Modeling*. John Wiley & Sons, 2008.
4. M. Blake, T. Weise, and S. Bleul. A graph-based approach to web services composition. In *Proceedings of the Applications and the Internet Symposium*, pages 183–189, 2005.

5. M. Blake, T. Weise, and S. Bleul. Wsc-2010: Web services composition and evaluation. In *Proceedings of the Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference*, pages 1–4, 2010.
6. G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1069–1075, 2005.
7. P. Chan and M. Luy. Dynamic web service composition: A new approach in building reliable web service. In *Proceedings of the Advanced Information Networking and Applications Conference*, pages 20–25, 2008.
8. W. Ching-Seh and I. Khoury. Tree-based search algorithm for web service composition in SaaS. In *Proc. of the International Conference, Information Technology: New Generations (ITNG)*, pages 132–138, 2012.
9. D. Chiu, S. Deshpande, G. Agrawal, and R. Li. Cost and accuracy sensitive dynamic workflow composition over grid. In *Proceedings of the 9th IEEE/ACM Int. Conf. on Grid Computing (GRID'08)*, pages 9–16, 2008.
10. H. Chun-hua, C. Xiao-hong, and L. Xi-ming. Dynamic services selection algorithm in web services composition supporting cross-enterprises collaboration. *Journal of Central South University of Technology*, 16(2):32–45, 2009.
11. D. B. Claro, P. Albers, and J. Hao. Selecting web services optimal composition. In *Proceedings of the 2nd Int. Workshop On Semantic And Dynamic Web Process*, pages 32–45, 2005.
12. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Polrola, and J. Skaruz. Harmonics - a tool for composing medical services. In *Proc. of ZEUS'12*, pages 25–33, 2012.
13. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Polrola, and M. Szreter. Web services composition - from ontology to plan by query. *Control & Cybernetics*, 40(2):315–336, 2011.
14. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Polrola, M. Szreter, and A. Zbrzezny. Planics - a web service compositon toolset. *Fundamenta Informaticae*, 112(1):47–71, 2011.
15. S. Eduardo, F. P. Luis, and S. V. Marten. Supporting dynamic service composition at runtime based on end-user requirements. In *Proceedings of the 1st Int. Workshop on User-generated Services (UGS2009)*, pages 464–471, 2009.
16. S. Eduardo, F. P. Luis, and S. V. Marten. Towards runtime discovery, selection and composition of semantic services. *Computer Communications*, 34(2):159–168, 2011.
17. K. Fujii and T. Suda. Semantics-based context-aware dynamic service composition. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–31, 2009.

18. I. Garibay, A. S. Wu, and O. Garibay. Emergence of genomic self-similarity in location independent representations. *Genetic Programming and Evolvable Machines*, 7(1):55–80, 2006.
19. Z. Jang, C. Shang, Q. Liu, and C. Zhao. A dynamic web services composition algorithm based on the combination of ant colony algorithm and genetic algorithm. *Journal of Computational Information Systems*, 6(8):2617–2622, 2010.
20. D. Kalyanmoy. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186:311–338, 2000.
21. F. Lecue, M. D. Penta, R. Esposito, and M. Villani. Optimizing qos-aware semantic web service composition. In *Proceedings of the 8th International Semantic Web Conference*, pages 375–391, 2009.
22. A. Niewiadomski, W. Penczek, and A. Polrola. SMT-based abstract planning in PlanICS ontology. *ICS PAS Reports*, 127:1–62, 2012.
23. J. A. Parejo, P. Fernandez, and A. R. Cortes. Qos-aware services composition using tabu search and hybrid genetic algorithms. *Actas de los Talleres de las Jornadas de Ingenieria del Software y Bases de Datos*, 2(1):55–66, 2008.
24. J. Peer. A pop-based replanning agent for automatic web service composition. *Lecture Notes in Computer Science*, 3532:189–198, 2005.
25. J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of SWSWPC’04*, pages 43–54, 2004.
26. T. Senivongse and N. Wongsawangpanich. Composing services of different granularity and varying QoS using genetic algorithm. In *Proceedings of the World Congress on Engineering and Computer Science*, 2011.
27. M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in daml-s. In *Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering*, pages 45–51, 2003.
28. A. S. Wu and R. Lindsay. A comparison of the fixed and floating building block representation in the genetic algorithm. *Evolutionary Computation*, 4(2):169–193, 1996.
29. Z. Zheng, Y. Zhang, and M. Lyu. Distributed QoS evaluation for real-world web services. In *Proc. of the Web Services (ICWS), 2010 IEEE International Conference on Web Services*, pages 83–90, 2010.