Evolutionary Algorithms for Abstract Planning

Jarosław Skaruz, Artur Niewiadomski, and Wojciech Penczek

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

Abstract. The paper presents a new approach based on evolutionary algorithms to an 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. 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. Experimental results are presented and compared with those obtained using an SMT-solver, showing that GA finds solutions for very large sets of service types in a reasonable and shorter time.

Keywords: Genetic algorithm · Web service composition · Abstract planning

1 Introduction

The number of web services available in the Internet has recently increased tremendously. 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 services need to be executed to this aim. The problem of finding such a composition is NP-hard [8] and well known as the Web Service Composition Problem (WSCP) [10].

There is a number of various approaches to solve WSCP [2]. Here, we follow the approach of the system Planics [3,4], which has been inspired by [1]. The main assumption is that all the web services in the domain of interest as well as the objects processed by them, can be strictly classified in a hierarchy of classes, organised in an ontology. Another key idea consists in having several stages of planning. The first phase deals with types (classes), while the second one - with

The research described in this paper has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.
the 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 the number of concrete services, which are taken into account.

This paper focuses on the abstract planning problem only. We propose a new approach based on an application of genetic algorithms. An individual of GA represents a multiset of service types and all the operations of GA are performed on this multiset. This feature of GA constitutes a great improvement in comparison to a linear representation of an individual as the algorithm does not need to care about the correct order of the service types represented. A linearization of an individual is generated only in order to compute its fitness value. An abstract plan is defined by a multiset of service types such that it has a linearization satisfying a user query. The algorithm stores each newly found abstract plan. In the subsequent iterations all similar individuals are ‘punished’ by decreasing their fitness value, proportionally to the similarity to all the abstract plans stored. To the best of our knowledge, the above approach is novel, and as our experiments show is also very promising.

As far as the related work is concerned, some approaches to WSCP are listed below. However, none of the existing algorithms for abstract planning uses genetic algorithms. The existing solutions to WSCP can be divided into several groups. Following [11] our approach belongs to AI planning methods, including also approaches based on: automata theory [12], Petri nets [14], theorem proving [13], and model checking [15]. The approach closest to ours is given in [7], where a genetic algorithm is used in one phase planning, which combines an abstract and a concrete one. Our idea of a multiset representation of a GA individual is not entirely new as it was already suggested in [5]. However, contrary to our approach, no linearization of a multiset is generated in order to compute the fitness value. In [9] the authors model a problem of non-coding DNA in biological systems in a form of genes, where their positions in an individual are not fixed, which leads to better experimental results. The constrained optimization problem, considered by us, was also studied in [6], where the penalty function does not need any parameter too, but ours is defined as a similarity measure.

The rest of the paper is organized as follows. Section 2 defines the abstract planning problem. Section 3 presents an application of GA to finding abstract plans. Section 4 discusses experimental results and provides a comparison with an SMT-based algorithm. The last section summarizes the results.

2 Abstract Planning Problem

This section introduces the Abstract Planning Phase (APP). APP makes intensive use of the service types and the object types defined in a given ontology. In what follows, let $S$ denote a set of all the service types defined in the ontology. A service type represents a set of web services with similar capabilities, while object types are used to represent data processed by the service types. The attributes are components of the object types. The ontology defines the inheritance relation, such that a subtype of some base object type retains all the attributes of
the base type, and optionally introduces some new attributes. The objects are instances of the object types. The values of the attributes of an object determine its state. A set of the objects in a certain state is called a world.

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. Its specification, as well as a service type specification, consists of three sets of objects: in, inout, and out, and two Boolean formulas, namely preCondition and postCondition (pre and post, for short). Pre is defined over attributes of the objects from in and inout, while post can involve also attributes of the objects from out. Since for APP there is no need to know the exact states of the objects, the values of the attributes are mapped to the two abstract values: set or null, denoting whether an attribute does have some value or it does not. An abstract world is a set of objects, which attributes have abstract values. In what follows we use the notions worlds and values instead of abstract worlds and abstract values, respectively.

A service type \( s \) is a pair of world sets \( (W_{in}^{pre}, W_{in}^{post}) \), called the input and the output worlds, respectively. That is, a service type is an interpretation of its 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. A user query \( q \) is a pair of world sets \( (W_{in}^{init}, W_{exp}^{exp}) \), called the initial and the expected worlds, respectively, defined similarly to the service types.

Example 1 (Service type). Consider an object type \( \text{Ware} \) containing the attributes \( \text{name}, \text{weight}, \text{owner}, \text{and location} \). Let \( \text{Transport} (T) \) be a service type able to deliver any instance of \( \text{Ware} \) to the requested destination, specified as: \( \text{in}_T = \text{out}_T = \emptyset, \text{inout}_T = \{ w : \text{Ware} \}, \text{pre}_T = \text{isNull}(w.\text{location}), \text{post}_T = \text{isNull}(w.\text{location}) \). Thus, \( W_{pre}^{T} \) is the set of worlds containing one instance of \( \text{Ware} \) with null location and any valuation of the remaining attributes, while in the worlds of \( W_{post}^{T} \) the location attribute is set.

Example 2 (User query). Assume that \( \text{Doghouse} \) is an object type extending \( \text{Ware} \). Consider that the user wants to obtain a doghouse. An example query \( q \) could be: \( \text{in}_q = \text{inout}_q = \emptyset, \text{out}_q = \{ d : \text{Doghouse} \}, \text{pre}_q = \text{true}, \text{post}_q = \text{isNull}(d.\text{location}) \). The interpretation of \( q \) is an empty initial world, and a set of expected worlds containing an instance of \( \text{Doghouse} \) with the attribute location set and any valuation of the remaining attributes.

World transformations. Assume we have two objects \( o_1 \) and \( o_2 \) of some worlds. 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 they agree on valuations of all common attributes. A world \( w_1 \) is compatible with a world \( w_2 \), if both of them contain the same number of objects and 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 \( |w| \). We say that 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 some input world of \( s \), restricted to the objects from \( in \),
- the objects from \( IN \), as well as the objects not involved in the transformation, do not change their states,
- \( w \) 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, \( w \) does not contain any of the objects from \( OU \), and \( |w'| = |w| + |OU| \).

We refer to a world transformation by a service type \( s \) also as an execution of \( s \).

Transformation sequences. Let \( seq = (s_1, \ldots, s_k) \) be a sequence of service types of length \( k \), and let \( w_0 \) and \( w_k \) be worlds, for some \( k \in \mathbb{N} \). 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 \( w_1, \ldots, w_{k-1}, \) such that \( w_{i-1} \xrightarrow{s_i} w_i \), for every \( i = 1, \ldots, k \).

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 \( w' \) is called the final world of \( seq \) while \( M_{seq} \) denotes the multiset of the service types \([s_1 + \cdots + s_k]\) 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 transforming \( w \) by \( seq \) is called the execution of \( seq \) in \( w \). Wo transformation sequences are called equivalent if they are built over the same multiset of service types.

User query solutions and abstract plans. Let \( seq = (s_1, \ldots, s_k) \) be a transformation sequence of length \( k \), and \( q = (W^q_{init}, W^q_{expr}) \) be a user query. We say that \( seq \) is a solution of the user query \( q \), if there are worlds \( w, w' \), such that \( w \xrightarrow{seq} w' \), \( w \in W^q_{init} \), and \( w' \in W^q_{expr} \). By \( QS_q \) we denote the set of all the solutions of \( q \). An abstract plan is defined as a set of all the solutions equivalent to some \( seq \in QS_q \), and is represented by the multiset of the service types \( M_{seq} \) for \( q \).

Example 3. 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, while Assembly is able to build a doghouse using nails and boards. If the user wants to obtain a doghouse, then there are several possibilities to achieve this goal. The shortest solution is the sequence \((S,T)\), which 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 and the second pair provides and delivers nails, which are finally assembled by \( A \) providing a doghouse. This solution constitutes another abstract plan represented by \([A + 2S + 2T]\). Note that there exists another equivalent solution, namely the sequence \((S,S,T,T,A)\).
3 Application of GA to Abstract Planning

The objective of GA is to find abstract plans (as many as possible) for a user query $q$. While GA maintains a population of individuals, each 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. If $seq_M$ is a solution of the user query $q$, then $M$ represents a new abstract plan.

The initial population of GA is generated randomly. As each gene of an individual models a service type, the number of all the genes is equal to the length of an abstract plan searched for. While we take advantage of the multiset representation, the order of the genes in an individual is irrelevant. 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. A linearization of an actual multiset is generated only in order to compute the fitness value of the corresponding individual.

Generating a sequence of service types $seq_M$ from a multiset $M$. The iterative procedure starts with an empty sequence and some initial world $w_0$, randomly selected at the start of GA from $W^\text{init}_M$. In the successive iterations a resulting sequence is built by removing from $M$ a service type $s$, which is able to transform a current world $w$. Then, the current world becomes the one obtained from the transformation of $w$ by $s$, and $s$ is appended to $seq_M$. 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 $seq_M$, the procedure returns also the length $l_M$ of the maximal executable prefix of $seq_M$, as well as the world $w_M$, obtained after transformation of $w$ by this prefix.

Example 4. Consider the multiset $M = [2A+S+T]$, where $A$, $S$, $T$ are the same service types as in Example 3, and let $w$ be an empty world. Only $S$ is able to transform the empty world, so it is appended to $seq_M$. Then, the current world contains a single instance of $Ware$ (or its subclass). In this case $T$ has to be chosen as the next service type, because $A$ needs at least two objects ($Nails$ and $Boards$), in order to be executed. Finally, two occurrences of $A$ are appended to the resulting sequence, and the procedure returns $seq_M = (S, T, A, A), l_M = 2$, and the world $w_M$ containing a $Ware$.

Fitness function. In order to evaluate an individual, its fitness value is calculated. To this aim the notion of a good service type is used. A service type is good, if it produces objects of types from the expected worlds, or of types from input worlds of other good service types. After an individual $M$ has been transformed to a sequence of service types $seq_M$, the fitness function takes the triple $(seq_M, l_M, w_M)$ and an expected world $w_q$ as the arguments, and is calculated according to (1):

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

2 Selected randomly from $W^{seq}_q$ at the start of GA.
Evolutionary Algorithms for Abstract Planning

\[ \text{fitness}(M) = \frac{f_{w_M} \cdot \delta + c_{w_M} \cdot \alpha + l_{M} \cdot \beta + g_{\text{seqM}} \cdot \gamma}{|w_q| \cdot \delta + |w_q| \cdot \alpha + k \cdot \beta + k \cdot \gamma} \]  

where: \( f_{w_M} = |w_{sub} | \) with \( w_{sub} \) being a maximal sub-world of \( w_M \) compatible with a sub-world of \( w_q \); \( c_{w_M} \) is the number of the objects from \( w_M \), which types are consistent with the types of objects from \( w_q \); \( g_{\text{seqM}} \) is the number of the good service types occurring in \( \text{seqM} \); \( k \) is the length of \( \text{seqM} \); and \( \alpha, \beta, \gamma \) are parameters of the fitness function. In all the experiments presented in Sect. 4 we use the following values: \( \alpha = 0.7, \beta = 0.1, \gamma = 0.2 \), and \( \delta = 0.1 \). These values of the parameters ensure that building a proper sequence of service types starts with a service type which produces an object required by a user. In the next steps, GA finds service types producing objects required by a user or needed as an input for the other service types in a given sequence.

After the first solution has been found, the aim of GA is to find other solutions remaining in the search space. On the other hand, each individual that represents a solution equivalent to some of the already found, should be eliminated. This is the task of the measure of similarity (the coefficient in (2)), between the currently rated individual and the plans found so far, used for modifying the fitness value of the individual. Let \( \text{Sol} \) denote a non-empty set of plans (in a form of multisets) found at some point of GA. Then, the modified fitness value of the individual \( M \) is calculated according to (2):

\[ \text{fitness}_{\text{Sol}}(M) = \left( 1.0 - \max \left( \left\{ \frac{|M \cap S|}{|M|} \bigg| S \in \text{Sol} \right\} \right) \right) \ast \text{fitness}(M) \]  

**Mutation operator.** Another original contributions of this paper consists in defining a mutation operator specialized for the discussed problem, which takes advantage of the good service type concept. So, 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. To this aim, one has to compute a set of all service types good for this sequence. If this set is not empty, then a randomly selected element of the set replaces the mutated gene. Notice that the mutation operator is not deterministic and it does not work in a greedy way.

### 4 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. Moreover, 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 \textit{in}, \textit{inout}, and \textit{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 2.8 GHz CPU and 8GB RAM.

Table 1 presents the summary of all the 18 experiments comparing the efficiency of our GA with an SMT-based planner [8]. 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 s, while SMT needs even over 500 s. 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.
Table 1. Experimental results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>1 64</td>
<td>2^{26}</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>—</td>
<td>4.28</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>1 42</td>
<td>2^{42}</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>—</td>
<td>7.76</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
<td>1 88</td>
<td>2^{88}</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>—</td>
<td>11.19</td>
</tr>
<tr>
<td>4</td>
<td>10 64</td>
<td>2^{26}</td>
<td>100 1 6</td>
<td>4.1 9</td>
<td>11</td>
<td>6</td>
<td>4.89</td>
<td>4.99</td>
<td>4.95</td>
</tr>
<tr>
<td>5</td>
<td>128</td>
<td>2^{42}</td>
<td>100 6</td>
<td>3.7 7</td>
<td>8</td>
<td>7.5</td>
<td>5.95</td>
<td>6.0</td>
<td>9.63</td>
</tr>
<tr>
<td>6</td>
<td>256</td>
<td>2^{88}</td>
<td>100 4</td>
<td>2.6 11</td>
<td>13</td>
<td>11</td>
<td>13.85</td>
<td>14.68</td>
<td>17.56</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>1 64 4</td>
<td>2^{14}</td>
<td>100 1</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>—</td>
<td>21.05</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>2^{43}</td>
<td>100 1</td>
<td>1</td>
<td>10</td>
<td>15</td>
<td>39.48</td>
<td>41.75</td>
<td>83.04</td>
</tr>
<tr>
<td>9</td>
<td>256</td>
<td>2^{27}</td>
<td>100 1</td>
<td>1</td>
<td>12</td>
<td>22</td>
<td>94.55</td>
<td>—</td>
<td>77.65</td>
</tr>
<tr>
<td>10</td>
<td>10 64</td>
<td>2^{24}</td>
<td>80 2</td>
<td>15 17</td>
<td>12</td>
<td>17.84</td>
<td>19.44</td>
<td>161.1</td>
<td>239.2</td>
</tr>
<tr>
<td>11</td>
<td>128</td>
<td>2^{43}</td>
<td>60 2</td>
<td>18 23</td>
<td>15</td>
<td>34.19</td>
<td>44.76</td>
<td>276.1</td>
<td>341.8</td>
</tr>
<tr>
<td>12</td>
<td>256</td>
<td>2^{72}</td>
<td>50 1</td>
<td>0.5 26</td>
<td>22</td>
<td>65.85</td>
<td>67.87</td>
<td>542.5</td>
<td>669.7</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>1 64 2</td>
<td>2^{72}</td>
<td>100 1</td>
<td>1</td>
<td>15</td>
<td>16</td>
<td>91.9</td>
<td>—</td>
</tr>
<tr>
<td>14</td>
<td>128</td>
<td>2^{94}</td>
<td>100 1</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>126.4</td>
<td>—</td>
<td>1141</td>
</tr>
<tr>
<td>15</td>
<td>256</td>
<td>2^{96}</td>
<td>90 1</td>
<td>0.9 29</td>
<td>34</td>
<td>213.1</td>
<td>—</td>
<td>&gt;1800</td>
<td>—</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>1 64 4</td>
<td>2^{10}</td>
<td>80 1</td>
<td>0.8</td>
<td>21</td>
<td>28</td>
<td>191.7</td>
<td>—</td>
</tr>
<tr>
<td>17</td>
<td>128</td>
<td>2^{105}</td>
<td>20 1</td>
<td>0.2 22</td>
<td>35</td>
<td>425.7</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>256</td>
<td>2^{120}</td>
<td>20 1</td>
<td>0.2 21</td>
<td>49</td>
<td>552.2</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1 presents the fitness value of the best individuals obtained in the experiments \textit{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. 1a) 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 \textit{exp4 - exp6} (Fig. 1b) we obtained a number of solutions within the first 20 iterations. In the experiments \textit{exp7 - exp9} (Fig. 1c) 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 \textit{exp10 - exp12} (Fig. 1d) 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 \textit{exp4 - exp6}, only in the experiment \textit{exp10} the next solution has been found.
5 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.
References