SMT versus Genetic Algorithms: Concrete Planning in the Planics Framework*

Extended Abstract

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

[1] ICS, Siedlce University, 3-Maja 54, 08-110 Siedlce, Poland
artur@ii.uph.edu.pl, jaroslaw.skaruz@uph.edu.pl
[2] ICS, Polish Academy of Sciences, Jana Kazimierza 5, 01-248 Warsaw, Poland
penczek@ipipan.waw.pl

Abstract. The paper deals with the concrete planning problem (CPP) – a stage of the Web Service Composition (WSC) in the PlanICS framework. A novel SMT-based approach to CPP is defined and its performance is compared to the standard Genetic Algorithm (GA) in the framework of the PlanICS system. The discussion of both the approaches is supported by extensive experimental results.

Keywords: Web Service Composition, SMT, GA, Concrete Planning

1 Introduction

The main concept of Service-Oriented Architecture (SOA) [2] consists in using independent components available via well-defined interfaces. Typically, a composition of web services need to be executed to realize the user objective. The problem of finding such a composition is hard and known as the WSC problem [2, 1, 11]. In this paper, we follow the approach of our system PlanICS [5, 6], which has been inspired by [1].

The main assumption in PlanICS is that all the web services in the domain of interest as well as the objects that are processed by the services, can be strictly classified in a hierarchy of *classes*, organised in an *ontology*. Another key idea is to divide the planning into several stages. The first phase deals with *classes of services*, where each class represents a set of real-world services, while the others work in the space of *concrete services*. The first stage produces an *abstract plan* composed of service classes [9]. Next, offers are retrieved by the offer collector (OC) (a module of PlanICS) and used by in a concrete planning (CP). As a result of CP a *concrete plan* is obtained, which is a sequence of offers satisfying predefined optimization criteria. Such an approach enables to reduce dramatically the number of web services to be considered, and inquired for offers. This paper deals with the concrete planning realised by SMT- and GA-based planners.

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

While CPP has been extensively studied in the literature as shown by numerous papers concerning an application of GA, the main contribution of our paper is an application of an SMT-based planner for finding optimal concrete plans. Such an approach based on SMT-solvers is quite promising and competitive comparing to applications of GA. The second contribution is a comparison of SMT-based approach performance with the results obtained from GA. While dealing with very large state spaces, an SMT-solver may be time demanding, but its advantage is demonstrated in finding always optimal concrete plans. Since a planner based on GA, being quite fast, may have difficulties in finding optimal concrete plans, we find both the approaches complementary.

In the last few years the CPP has been extensively studied in the literature. In [4] a simple GA was used to obtain a good quality concrete plan. In [12] CPP was transformed to a multicriteria optimization problem and GA was used to find a concrete plan. However, the authors present the experiments on a relatively small search space that could not provide valuable conclusions. Our paper fills the gap by presenting the results that allow to examine scalability of the algorithms and their efficiency when the large search space is considered.

Most of the applications of SMT in the domain of WSC is related to the automatic verification and testing. For example, a message race detection problem is investigated in [7], [3] deals with a service substitutability problem, while [8] exploits SMT to verification of WS-BPEL specifications against business rules. However, to the best of our knowledge, there are no other approaches dealing with SMT-solvers as an engine to WSC.

The rest of the paper is organized as follows. Sect. 2 deals with CPP in PlanICS. Sect. 3 presents our SMT-based approach to solve CPP. Sect. 4 discusses experimental results compared to results of the standard GA, while the last section summarizes the results.

2 Concrete Planning Problem

This section defines CPP as the third stage of the WSC in PlanICS framework and provides the basic definitions. We introduce the main ideas behind PlanICS and define CPP as a constrained optimization problem. PlanICS is a system implementing our original approach which solves the composition problem in clearly separated stages. An ontology, managed by the *ontology provider*, contains a system of *classes* describing the types of the services as well as the types of the objects they process [10]. A class consists of a unique name and a set of the attributes. By an *object* we mean an instance of a class. Below, we present a simplified ontology, used further as a running example.

Example 1 (Ontology). Consider a simple ontology describing a fragment of some financial market consisting of service types, inheriting from the class $Investment$, representing various types of financial instruments, and three object types: $Money$ having the attribute *amount*, $Transaction$ having the two attributes *amount* and $profit$, and $Charge$ having the attribute fee. Suppose that each investment

service takes m - an instance of $Money$ as input, produces t and c - instances of $Transaction$ and $Charge$, and updates the amount of money remaining after the operation, i.e, the attribute $m.amount$.

Two fundamental concepts of PlanICS are *worlds* and *world transformations*.

Definition 1 (World, Abstract World). *Let $\mathcal{D} = \mathbb{Z} \cup \mathbb{R} \cup \mathbb{A}$, where $\mathbb{Z}$ is the set of integers, $\mathbb{R}$ is the set of real numbers, and $\mathbb{A} = \{\mathbf{set}, \mathbf{null}\}$ is the set of* abstract values*. Let $\mathcal{O}$ be the set of all objects, $\mathcal{A}$ be the set of all attributes, and $attr : \mathcal{O} \mapsto 2^{\mathcal{A}}$ be the function returning the attributes of an object. A* world *is a pair (O, val), where $O \subseteq \mathcal{O}$ is a set of objects and $val : O \times attr(O) \mapsto \mathcal{D}$ is a valuation function, which to every attribute of the objects from O assigns a value from a respective domain or* **null***, if the attribute does not have a value. A world is* abstract *if all its object attributes have values from $\mathbb{A}$.*

Since the main part of this paper deals with an optimization problem, the domains under consideration are the integer and the real numbers[3]. PlanICS uses a state-based approach in which the worlds represent 'snapshots' of the reality, while the services transform them. A *transformation* of a world w by a service of type s into a world w', denoted by $w \xrightarrow{s} w'$, consists in processing a subset of w, by changing values of object attributes and/or adding new objects, according to the specification of s [10]. Often, a world w can be transformed by s in more than one way. For example when w contains multiple objects of the same type and one can designate more than one subset which can be processed by s. Thus, we define a *transformation context* cx as a mapping from the objects of the input and output of s to the objects of w. The transformation of w by s in the context cx to w' is denoted by $w \xrightarrow{s,cx} w'$ [9].

The user expresses a goal by a *query*, referring to objects and their attributes, and adding constraints while defining *initial worlds* to start with and *expected worlds* to be reached. Composition is thus understood as searching for a set of services capable to process certain states in a desired way, that is, transforming a subset of an initial world into a superset of an expected world (called a *final world*). This is obtained by executing services according to a *plan*.

A specification of a user query consists of the following components: three sets of objects IN, IO, and OU, two boolean formulas Pre (over $IN \cup IO$) and $Post$ (over $IN \cup IO \cup OU$) specifying the initial and the expected worlds, resp., a set of aggregate conditions and a set of quality expressions, to be defined later. The objects of IN are read-only, these of IO can change values of their attributes, and the objects of OU are produced in subsequent transformations. The Pre and $Post$ formulas define two families of valuation functions V_{Pre} and V_{Post}, determining values of objects from the initial and the expected worlds, resp. A *set of worlds* is defined by a pair composed of a set of objects and a family of valuation functions. In general, there are three main cases, when Pre or $Post$ formula defines a family of valuation functions instead of a single function.

[3] Note that other types of values used in PlanICS framework, like strings, dates, and Boolean values can be easily coded by integers.

That is, when a formula contains: (i) an alternative, (ii) constraints that can be satisfied by more than one valuation, or (iii) the formula does not specify values of some attributes. In order to define a user query in a formal way, we need to define the aggregate conditions and the quality expressions which, contrary to the *Pre* and *Post* formulas, are evaluated over final worlds, so they can take into account also objects not foreseen by the user, but created as by-products of the transformations leading to the final worlds.

Definition 2 (Aggregate conditions, Quality expressions). *A quality expression is a tuple* $(cl, sel, form, type)$*, where cl is an object type (a class from the ontology), sel is a boolean formula over attributes belonging to cl,* $form$ *is a real valued expression (built using standard arithmetic operators, like addition, subtraction, multiplication and division) over attributes of class cl, and* $type \in \{Sum, Min, Max\}$*. An aggregate condition is a tuple* $(cl, sel, form, type, \sim, lim)$*, where the first four components are defined as above,* $\sim \in \{<, \leq, =, \neq, >, \geq\}$ *is a comparison operator, and* $lim \in \mathbb{R}$*. A set of aggregate conditions is denoted by Aggr, while a set of quality expressions is denoted by Qual.*

The purpose of *Qual* is to specify criteria of the best plan, while *Aggr* is used in order to add sophisticated restrictions on the resulting plan. Their interpretation is the following. In order to evaluate a single aggregate condition or a quality expression, first we need to separate a subset of a final world containing the objects of type *cl* only. Next, we restrict this subset to the objects satisfying *sel* condition. Then, for each object from the remaining set we compute the value of $form$ expression. Finally, the aggregation function *type* is applied to the obtained set of values and as a result we get a single (real) value. In the case of an aggregate condition, the obtained value is compared with lim constant, using the provided operator $\sim$, and as a result we get a boolean value.

Example 2 (Query specification). Consider the ontology from Example 1. Assume that the user would like to invest up to \$100 in three financial instruments, but he wants to locate more than \$50 in two investments. The above is expressed by: $IN = \emptyset$, $IO = \{m : Money\}$, $OU = \{t_1, t_2, t_3 : Transaction\}$, $Pre = (m.amount \leq 100)$, and $Post = (t_1.amount + t_2.amount > 50)$. The best plan is clearly this which is the most profitable, i.e., the user wants to maximize the sum of profits. Moreover, he wants to use only services of handling fees less than \$3. The above conditions are expressed by the following aggregate condition and the quality expression: $Aggr = \{(Charge, true, fee, Max, <, 3)\}$, and $Qual = \{(Transaction, true, profit, Sum)\}$.

Formally, a user query is defined as follows:

Definition 3 (User query). *A user query is a tuple* $(W^I, W^E, Aggr, Qual)$*, where* $W^I = (IN \cup IO, V_{Pre})$ *and* $W^E = (IN \cup IO \cup OU, V_{Post})$ *are sets of initial and expected worlds, respectively, Aggr is a set of aggregate conditions, and Qual is a set of quality expressions.*

In the first stage of composition an *abstract planner* matches services at the level of input/output types and the abstract values. The result of this stage is a

Context Abstract Plan (CAP, for short), to be defined after introducing auxiliary definitions. At this planning stage it is enough to know if an attribute does have a value, so we abstract from the concrete values of the object attributes [9], using the following definition.

Definition 4 (World correspondence). *Let $w = (O, val)$ be a world and $w' = (O, val')$ be an abstract world. We say that w'* corresponds *to w iff for every $o \in O$ and for every $a \in attr(o)$* $val'(o,a) = \begin{cases} \textbf{set}, & \text{for } val(o,a) \neq \textbf{null}, \\ \textbf{null}, & \text{for } val(o,a) = \textbf{null}. \end{cases}$

In order to compose services, we define the transformation sequences.

Definition 5 (Transformation sequence). *Let k be a natural number and $seq = \big((s_1, cx_1), \ldots, (s_k, cx_k)\big)$ be a sequence of length k, where s_i is a service type and cx_i is a transformation context for $i = 1, \ldots, k$. We say that a world w_0* is transformed by *seq into a world w_k iff there exists a sequence of worlds $(w_1, w_2, \ldots, w_{k-1})$ such that $\forall_{1 \leq i \leq k}\ w_{i-1} \xrightarrow{s_i, cx_i} w_i$. A sequence seq is called a* transformation sequence*, if there are two worlds w, w' such that w is transformed by seq into w'. The world w' is called a* final world *of seq.*

Finally, we are in a position to define the result of the abstract planning phase.

Definition 6 (Abstract solution, CAP). *Given a transformation sequence seq and a user query $q = (W^I, W^E, Aggr, Qual)$. We say that seq is an abstract solution for q iff for some $w_0 \in W^I$, $w_k \in W^E$, there are abstract worlds w_I, w_F, such that w_I corresponds to w_0 and w_F corresponds to w_k and w_I is transformed by seq into w_F. A CAP for a query q is a pair $CAP_q = (seq, w_F)$, where seq is an abstract solution for q and w_F is a final world of seq.*

Thus, each CAP (seq, w_F) contains information on the service types, the context mappings, and a final world of *seq*. Note that using CAP, the ontology, and the user query we are able to reproduce all the worlds of the transformation sequence. A sequence *seq* is just a representative of a class of equivalent sequences [9, 10].

Collecting offers. In the second planning stage CAP is used by an *offer collector* (OC), i.e., a tool which in cooperation with the service registry queries real-world services. The service registry keeps an evidence of real-world web services, registered accordingly to the service type system. During the registration the service provider defines a mapping between input/output data of the real-world service and the object attributes processed by the declared service type. OC communicates with the real-world services of types present in a CAP, sending the constraints on the data, which can potentially be sent to the service in an inquiry, and on the data expected to be received in an offer in order to keep on building a potential plan. Usually, each service type represents a set of real-world services. Moreover, querying a single service can result in a number of offers. Thus, we define an offer set as a result of the second planning stage.

Definition 7 (Offer, Offer set). *Assume that the n-th instance of a service type from a CAP processes some number of objects having in total m attributes. A single offer collected by OC is a vector $P = [v_1, v_2, \ldots, v_m]$, where v_j is a value of a single object attribute from the n-th intermediate world of the CAP.*

An offer set O^n is a $k \times m$ matrix, where each row corresponds to a single offer and k is the number of offers in the set. Thus, the element $o^n_{i,j}$ is the j-th value of the i-th offer collected from the n-th service type instance from the CAP.

Example 3 (Offer, Offer sets). Consider the user query from Example 2 and an exemplary CAP consisting of three instances of $Investment$ service type. A single offer collected by OC is a vector $[v_1, v_2, v_3, v_4, v_5]$, where v_1 corresponds to $m.amount$, v_2 to $t.amount$, v_3 to $t.profit$, and v_4 to $c.fee$. Since the attribute $m.amount$ is updated during the transformation, the offers should contain values from the world before and after the transformation. Thus v_5 stands for the value of $m.amount$ after modification. Assuming that instances of $Investment$ return k_1, k_2, and k_3 offers in response to subsequent inquiries, we obtain three offer sets: O^1, O^2, and O^3, where O^i is a $k_i \times 5$ matrix of offer values.

At the moment we develop two implementations of OC realizing the "simple", and the "intelligent" concept. The goal of the first approach is to rule out the offers violating simple constraints from the user query. An intelligent OC, taking advantage of an inference mechanism, a symbolic computation engine, and the semantic knowledge from the ontology, aims at discovering more sophisticated dependencies between offers and use them while collecting offers. Regardless of the approach chosen, every implementation of OC should satisfy some common requirements: *a*) the ability of a reconstruction of the intermediate worlds from a CAP, *b*) returning offer sets corresponding to the objects processed by the service types instances from a CAP, filled with the values acquired from real-world services, *c*) propagating the values and constraints present in the user query and returning them as expressions over offer sets, *d*) capturing the dependencies between the values of object attributes from the worlds of a CAP and returning them as expressions over offer sets, *e*) translating the set of quality expressions specified as a part of the query to a scalar function defined over offer sets, being the sum of all quality constraints.

In the third planning stage, the offers are searched by a *concrete planner* in order to find the best solution satisfying all constraints and maximising a quality function. Thus, we can formulate CPP as a constrained optimization problem.

Definition 8 (CPP). *Let n be the length of CAP and let $\mathbb{O} = (O^1, \ldots, O^n)$ be the vector of offer sets collected by OC such that for every $i = 1, \ldots, n$*

$O^i = \begin{bmatrix} o^i_{1,1} & \cdots & o^i_{1,m_i} \\ \vdots & \ddots & \vdots \\ o^i_{k_i,1} & \cdots & o^i_{k_i,m_i} \end{bmatrix}$*, and the j-th row of O^i is denoted by P^i_j. Let $\mathbb{P}$ denote the set of all possible sequences $(P^1_{j_1}, \ldots, P^n_{j_n})$, such that $j_i \in \{1, \ldots, k_i\}$ and $i \in \{1, \ldots, n\}$. The Concrete Planning Problem is defined as:*

$$max\{Q(S) \mid S \in \mathbb{P}\} \text{ subject to } \mathbb{C}(S), \tag{1}$$

where $Q : \mathbb{P} \mapsto \mathbb{R}$ is an objective function defined as the sum of all quality constraints and $\mathbb{C}(S) = \{C_j(S) \mid j = 1, \ldots, c \text{ for } c \in \mathbb{N}\}$, where $S \in \mathbb{P}$, is a set of constraints to be satisfied.

A solution of CPP consists in selecting one offer from each offer set such that all constraints are satisfied and the value of the objective function is maximized.

Theorem 1. *The concrete planning problem (CPP) is NP-hard.*

Proof. See Appendix.

3 Concrete Planning using SMT

This section deals with our novel application of SMT to CPP viewed as a constrained optimization problem. The idea is to encode CPP as an SMT formula such that there is a solution to CPP iff the formula is satisfiable. First, a set $\mathcal{V}$ of all necessary SMT-variables (for simplicity called just *variables*) is allocated:

- $\mathbf{q}$ - for storing the subsequent values of the quality function found,
- $\mathbf{oid}^i$, where $i = 1 \ldots n$ and n is the length of the abstract plan. These variables are needed to store the identifiers of offers constituting a solution. A single $\mathbf{oid}^i$ variable takes a value between 1 and k_i.
- $\mathbf{o}^i_j$, where $i = 1 \ldots n$, $j = 1 \ldots m_i$, and m_i is the number of offer values in the i-th offer set. We use them to encode the values of S, i.e., the values from the offers chosen as a solution. From each offer set O^i we extract the subset R^i of offer values which are present in the constraint set and in the quality function, and we allocate only the variables relevant for the plan.

Next, using the variables from $\mathcal{V}$, the offer values from the offer sets $\mathbb{O} = (O^1, \ldots, O^n)$ are encoded as the formula

$$ofr(\mathbb{O}, \mathcal{V}) = \bigwedge_{i=1}^{n} \bigvee_{d=1}^{k_i} \Big(\mathbf{oid}^i = d \ \wedge \bigwedge_{o^i_{d,j} \in R^i} \mathbf{o}^i_j = o^i_{d,j} \Big). \tag{2}$$

Then, the conjunction of all constraints is encoded as the formula $ctr\big(\mathbb{C}(S), \mathcal{V}\big)$, and the objective function as the formula $qual\big(Q(S), \mathcal{V}\big)$. For convenience its value is bound with the variable $\mathbf{q}$ by $\mathbf{q} = qual\big(Q(S), \mathcal{V}\big)$. Thus, the formula encoding the solutions of CPP is as follows:

$$cpp\big(\mathbb{O}, Q(S), \mathbb{C}(S), \mathcal{V}\big) = ofr\big(\mathbb{O}, \mathcal{V}\big) \wedge ctr\big(\mathbb{C}(S), \mathcal{V}\big) \wedge \mathbf{q} = qual\big(Q(S), \mathcal{V}\big) \tag{3}$$

The maximal value of $\mathbf{q}$ is searched using the $SMTsearch$ procedure presented in Alg. 1 adapting the *binary search* method. The *assumptions* mechanism of an SMT-solver is exploited, which consists in checking satisfiability of an SMT-formula assuming that a set of boolean conditions are satisfied. In every iteration the searched interval is divided in half and, since the objective function is to be

Procedure SMTsearch($cpp\big(\mathbb{O}, Q(S), \mathbb{C}(S), \mathcal{V}\big)$, δ, min, max)
Input: encoded formula, accuracy, estimated min. and max. value of $Q(S)$
Result: the maximal value of $\mathbf{q}$ with an accuracy of δ

begin
  $pivot \leftarrow (min + max)/2$; $a_1 \leftarrow (\mathbf{q} > pivot)$; $i \leftarrow 1$; $result \leftarrow null$;
  $A \leftarrow \{a_1\}$; `// a single assumption` a_1 `in the assumption set` A
  while $(|max - min| > \delta)$ **do**
    if $checkSat(cpp\big(\mathbb{O}, Q(S), \mathbb{C}(S), \mathcal{V}\big), A)$ **then**
      $i \leftarrow i + 1$; $result \leftarrow \mathbf{q}$; $min \leftarrow \mathbf{q}$; $pivot \leftarrow (min + max)/2$;
    else
      $A \leftarrow (A \setminus \{a_i\}) \cup \{\neg a_i\}$; `// replace` a_i `by` $\neg a_i$ `in` A
      $max \leftarrow pivot$; $pivot \leftarrow (min + max)/2$; $i \leftarrow i + 1$;
    end
    $a_i \leftarrow (\mathbf{q} > pivot)$; $A \leftarrow A \cup \{a_i\}$;
  end
  return $result$
end

Algorithm 1: Pseudocode of the SMT-based CPP algorithm

maximized, a solution of value greater than a half ($pivot$) is searched. To this end the whole formula is checked for satisfiability under the assumption ($\mathbf{q} > pivot$). If there is a solution, then its value becomes min. Otherwise, the searched value is less or equal $pivot$, the last assumption is replaced by its negation, and $pivot$ value is assigned to max. Then, a new $pivot$ value is computed and the algorithm iterates again, while the length of the searched interval is greater than δ.

4 Experimental Results

In this section we present the results of the experiments performed using the Z3 SMT-solver running on a standard PC equipped with 2GHz CPU and 8GB RAM. Since Genetic Algorithms are widely used in many optimization problems, we compare the efficiency of our new SMT-based approach with the results obtained using the standard GA, which we have implemented to this aim.

Implementation of GA. The only non-standard elements of our GA are the concrete plan encoding scheme and the computation of the fitness function. An individual is a sequence of indices of the offers chosen from the consecutive offer sets. The fitness value of an individual is the sum of the optimization objective and the ratio of the number of the satisfied constraints to the number of all constraints (see Def. 8), multiplied by some constant β:

$$fitness(Ind) = Q(S_{Ind}) + \beta \cdot \frac{|sat(\mathbb{C}\big(S_{Ind}\big))|}{c}, \tag{4}$$

where Ind stands for an individual, S_{Ind} is a sequence of the offer values corresponding to Ind, $sat\big(\mathbb{C}(S_{Ind})\big)$ is a set of the constraints satisfied by a candidate

solution represented by *Ind*, and c is the number of all constraints. The role of β is to reduce both of the sum components to the same order of magnitude and to control the impact of the components on the final result.

Experiments. In each of the experiments we use different optimization objectives and constraints, and compare the obtained results. Equation 5 presents the objectives Q_1, Q_2, Q_3 used in the experiments 1-5, while the constraints are combinations of $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_3$ defined by Equation 6. In the experiments 6 and 7, we use the constraints and the objective function of our working example.

$$Q_1 = \sum_{i=1}^{n} o^i_{j_i,1}, \qquad Q_2 = \sum_{i=1}^{n} o^i_{j_i,2}, \qquad Q_3 = \sum_{i=1}^{n} (o^i_{j_i,1} + o^i_{j_i,2}), \tag{5}$$

$$\mathbb{C}_1 = \{(o^i_{j_i,1} < o^{i+1}_{j_{i+1},1})\}, \quad \mathbb{C}_2 = \{(o^i_{j_i,2} < o^{i+1}_{j_{i+1},2})\}, \quad \mathbb{C}_3 = \{(o^i_{j_i,2} = o^{i+1}_{j_{i+1},2})\} \text{ for } i = 1, \ldots, n-1. \tag{6}$$

In all the experiments sets of offers generated randomly by our Offer Generator (OG) have been used. The values have been uniformly distributed in the range between 0 and 100, but the α parameter has been introduced, which specifies a percentage of values below 33.3, in order to get different distributions of high quality concrete plans. The following values of the GA parameters have been used: the number of the individuals equals to 1000, the probability of mutation equals to 0.5%, the probability of the one-point crossover operator equals to 95%, and the algorithm was run 100 times for each setup. As to the SMT-based algorithm, the 500 sec. time-out has been set. Each experiment has been repeated only 10 times as the run-times obtained have been very similar, and the quality values have been the same each time.

In Experiment 1 Q_1 has been used as an optimization objective and $\mathbb{C}_1$ as a set of constraints. In all experiments we have tested instances with 5, 10, and 15 offer sets, containing 256, 512, and 1024 offers each. Two sets of offers with $\alpha = 5\%$ and $\alpha = 40\%$ have been generated. The results are presented in Table 1. The SMT-based planner always returns optimum, while GA, as a non-deterministic algorithm, finds optimum in at most 20% cases, for instances with 5 service types. All solutions are obtained within 0.25 to nearly 3 seconds. Comparing quality we can observe that the difference between the SMT-solver and GA ranges from 0.5% up to about 31% for instances with 15 service types.

In Experiment 2 the constraints remain the same ($\mathbb{C}_1$), the objective function is similar (Q_2), but we maximize the sum of other values than these used in the constraints. Table 2 presents the impact of these changes on the SMT-based planner. In comparison to the results of Exp. 1, the runtime of SMT-solver increases.The biggest difference can be noticed for plans of length 15, however, as it follows from the probability results, these are also hard to find for GA. On the other hand, the power of SMT is in the ability to take advantage of the constraints in order to reduce the search space. The results suggest that working with constraints related to different variables than these used in the objective function leads to longer runtimes of the SMT-solver.

Table 1. The results of Experiment 1: left $\alpha = 40\%$, right $\alpha = 5\%$.

			SMT		GA				SMT		GA			
Sp.	**n**	**Offs**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Bs.**	**Pr.**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Bs.**	**Pr.**
2^{45}	5	512	0.28	485	0.41	479.23	18	100	0.36	490	0.42	484.95	13	100
2^{50}		1024	0.52	490	0.86	481.95	4	100	0.52	490	0.76	484.9	9	100
2^{80}	10	256	0.51	920	0.47	720.21	0	92	0.47	955	0.47	794.64	0	91
2^{90}		512	0.68	955	0.83	797.97		91	0.72	955	0.83	824.96		97
2^{100}		1024	1.27	955	1.59	802.44		100	1.33	955	1.59	853.56		98
2^{120}	15	256	0.73	1350	0.9	929.3		10	0.95	1388	0.76	1102.07		14
2^{135}		512	1.49	1377	1.47	998.8		15	1.28	1395	1.35	1092.33		12
2^{150}		1024	2.06	1395	2.78	1027.43		16	2.09	1395	2.53	1086.85		14

Table 2. The results of Experiment 2: left $\alpha = 40\%$, right $\alpha = 5\%$.

			SMT		GA				SMT		GA			
Sp.	**n**	**Offs**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Bs.**	**Pr.**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Bs.**	**Pr.**
2^{45}	5	512	0.87	499	0.37	493.62	4	100	0.50	499	0.39	498	17	100
2^{50}		1024	1.55	499	0.72	495.91	9	100	0.65	499	0.67	498.72	28	100
2^{80}	10	256	4.47	979	0.46	934.69	2	89	2.19	994	0.45	957.78	1	88
2^{90}		512	3.44	992	0.83	923.62	1	89	2.76	996	0.83	967.37		90
2^{100}		1024	6.02	995	1.53	949.09		93	3.34	998	1.57	959.55		98
2^{120}	15	256	249.46	1443	0.74	1269.87	0	8	100.26	1475	0.73	1416	0	9
2^{135}		512	425.32	1467	1.5	1322.92		12	97.20	1489	1.35	1362.28		7
2^{150}		1024	57.74	1493	2.67	1300.58		12	63.94	1495	2.49	1398.38		18

Table 3. The results of Experiment 3 (left) and Experiment 4 (right).

			SMT		GA			SMT		GA		
Sp.	**n**	**Offs**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Pr.**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Pr.**
2^{45}	5	512	7.73	924	0.36	865.8	100	1.59	841	0.32	754.51	100
2^{50}		1024	3.95	947	0.61	892.13	100	1.41	904	0.62	781.3	100
2^{80}	10	256	> 500	1633*	0.4	1464.64	93	150.58	1207	0.4	1025.5	2
2^{90}		512	215.13	1803	0.7	1479.98	86	24.09	1562	0.69	1187.25	4
2^{100}		1024	243.24	1862	1.29	1535.52	96	345.57	1655	1.34	1211.72	11
2^{120}	15	256	> 500	2448*	0.64	2067.45	11	> 500	1550*	—		0
2^{135}		512		2291*	1.15	2148.08	12		1567*			
2^{150}		1024		2630*	2.11	2134.35	17		2270*			

Table 4. The results of Exp. 5 (left), Exp. 6 (center), and Exp. 7 (right).

			SMT		GA			SMT		GA			SMT	
Sp.	**n**	**Offs**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Pr.**	**t[s]**	**Q**	**t[s]**	**AvgQ**	**Pr.**	**t[s]**	**Q**
2^{45}	5	512	0.54	752	0.34	356.6	14	3.63	393	0.32	266.26	30	0.41	5
2^{50}		1024	0.85	866	0.64	420.2	10	3.48	469	0.6	274.86	30	0.68	10
2^{80}	10	256	0.73	noSol	—		0	35.24	683	0.4	289	8	0.32	noSol
2^{90}		512	5.01	780				159.09	753	0.68	399.22	18	1.18	12
2^{100}		1024	3.51	1508				143.07	888	1.3	378.5	24	2.78	20
2^{120}	15	256	1.10	noSol				> 500	942*	0.64	446	1	1.00	noSol
2^{135}		512	10.43	1164					767*	1.14	536	2	5.54	18
2^{150}		1024	40.82	1287					755*	2.08	481	5	16.97	29

In order to discover the limitations of both the planners, in Experiments 3, 4, and 5 we use the harder set of data, i.e., the offers generated with $\alpha = 40\%$. We examine how the approaches deal with the "sum of sums" in the optimization function, and thus in Exp. 3 we use Q_3 as the optimization objective and $\mathbb{C}_1$ as the set of constraints. Moreover, we aim at confirming our conjecture on behaviour of the SMT-solver in presence of a larger number of constraints. Thus, in Exp. 4 we use Q_3 as the optimization objective and $\mathbb{C}_1 \cup \mathbb{C}_2$ as the set of constraints. Table 3 presents the results.

Comparing the results of Exp. 2 and 3 one can notice that the more complicated objective function has almost no influence on the runtime and the probability of finding solutions by GA. On the other hand, the instances from Exp. 3 seem to be a bit harder for GA because the quality difference between SMT and GA is greater and ranges from 3.4% up to about 19%, and furthermore, GA could not find an optimal solution. The results of our SMT-based approach indicate that the constraints used in Exp. 3 are too weak to significantly bound the search space. We were unable to find the optimal solution in four cases.

In Exp. 4 we use two times more constraints than in Exp. 3. Firstly, we found the limit beyond which an application of GA is pointless. In Exp. 4 we use $2 \cdot (n-1)$ constraints. For plans of length 10 and for 18 constraints GA barely finds a few solutions quality of which differ from optimum by 12% to almost 27%. Secondly, adding more constraints improves slightly the efficiency of an SMT-solver. However, not only the number of constraints is important, but also their influence on the number of existing solutions in the search space. We prove it by choosing $\mathbb{C}_1 \cup \mathbb{C}_3$ as the set of constraints (i.e., we change the half of constraints from "less than" to "equal") and running Exp. 5. The results are in Table 4. The SMT-runtime decreases tremendously, as well as quality and the probability of finding a solution by GA. Now, GA barely finds solutions of length 5 quality of which differ from optimum by 13.7% up to almost 53%.

Experiments 6 and 7 are based on our working example and have been performed on datasets with $\alpha = 90\%$. In Exp. 6 we used 6, 11, and 14 constraints for plans of length 5, 10, and 15, respectively. The quality of solutions found by GA is worse by about 30% to almost 58% than the ones found by the SMT-solver. Moreover, the solutions have been found with a low probability. Unfortunately, the runtimes of SMT-based planner are quite long in this case. However, using a set of 11, 21, and 29 constraints for plans of length 5, 10, and 15, respectively, which significantly reduces the number of solutions existing in the search space, we obtain a very nice behaviour of our SMT-based planner in Exp. 7. Table 4 presents the results, where *noSol* means that there is no solution at all.

5 Conclusions

In the paper we have presented the concrete planning in the Planics framework, its reduction to the constrained optimization problem, and a new SMT-based approach to solve it. The experimental results, compared with results of the standard GA, present advantages and disadvantages of both the approaches. The

most important feature of the SMT-based planner is its ability of finding always the optimal solution, provided that it is enough time and memory. In contrast, GA finds sometimes the optimal solution of length at most 5, but it consumes less time and memory. The ability of GA to find a concrete plan depends on the number of constraints. The more optimization constraints the smaller probability of finding a concrete plan. These drawbacks of GA are not common to our SMT-based approach. Moreover, our experiments show that a large number of constraints helps the SMT-solver to reduce the search space and to find the optimal solution faster. Our experimental results show that an application of the SMT-based method to solve CPP is promising and valuable against the well known GA-based approach. Overall, both the approaches are complementary and behave differently depending upon a particular problem instance.

References

1. S. Ambroszkiewicz. Entish: A language for describing data processing in open distributed systems. *Fundam. Inform.*, 60(1-4):41–66, 2004.
2. M. Bell. *Introduction to Service-Oriented Modeling.* John Wiley & Sons, 2008.
3. M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi. SMT-based verification of LTL specification with integer constraints and its application to runtime checking of service substitutability. In *SEFM*, pages 244–254, 2010.
4. 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.
5. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, and J. Skaruz. HarmonICS - a tool for composing medical services. In *ZEUS*, pages 25–33, 2012.
6. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, and A. Zbrzezny. PlanICS - a web service compositon toolset. *Fundam. Inform.*, 112(1):47–71, 2011.
7. M. Elwakil, Z. Yang, L. Wang, and 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, pages 182–194. Springer, 2010.
8. G. Monakova, O. Kopp, F. Leymann, S. Moser, and K. Schäfers. Verifying business rules using an SMT solver for BPEL processes. In *BPSC*, pages 81–94, 2009.
9. A. Niewiadomski and W. Penczek. Towards SMT-based Abstract Planning in PlanICS Ontology. In *Proc. of KEOD 2013 – International Conference on Knowledge Engineering and Ontology Development*, 2013 (to appear).
10. A. Niewiadomski, W. Penczek, and A. Półrola. Abstract Planning in PlanICS Ontology. An SMT-based Approach. Technical Report 1027, ICS PAS, 2012.
11. J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of SWSWPC'04*, volume 3387 of *LNCS*, pages 43–54. Springer, 2004.
12. Y. Wu and X. Wang. Applying multi-objective genetic algorithms to qos-aware web service global selection. *Advances in Information Sciences and Service Sciences*, 3(11):134–144, 2011.

Appendix

Proof (NP-hardness of CPP). We show that concrete planning problem is NP-hard by showing the linear reduction of 3-SAT problem to CPP. Consider a set of propositional variables $\mathcal{PV}$ and 3-CNF formula $\varphi = c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where $c_i = (x_i^1 \vee x_i^2 \vee x_i^3)$, $x_i^j = p$ or $x_i^j = \neg p$, for every $p \in \mathcal{PV}$, $i = 1 \ldots n$ and $j = 1 \ldots 3$. We encode the satisfiability problem of φ as a Concrete Planning Problem CPP as follows.

We take a Context Abstract Plan (CAP) of length n, and n offer sets. Each offer contains 3 values from the set $\{0, 1\}$, and each value corresponds to a single propositional variable used in i-th clause. Each offer set contains all the possible combinations of offer values (8 offers per set), that is, each offer set is an 8×3 matrix. Thus, $\mathbb{P}$ is the set of all possible binary sequences of length $3n$.

We transform the formula φ to a set of constraints $\mathbb{C}$ in such a way that every clause c_i became a single constraint, where $x_i^j = p$ is encoded as $o^i_{k_i,j} = 1$ and $x_i^j = \neg p$ as $o^i_{k_i,j} = 0$, for $k_i = 1 \ldots 8$ and $p \in \mathcal{PV}$. Moreover, for every propositional variable p occurring in φ we take two subsets of offer variables P_p and N_p, which encode p and $\neg p$, respectively: $P_p = \{o^i_{k_i,j} \mid \text{for every } i, j \text{ such that } x_i^j = p\}$ and $N_p = \{o^i_{k_i,j} \mid \text{for every } i, j \text{ such that } x_i^j = \neg p\}$. Now, for each non-empty set X_p, where $X \in \{P, N\}$ and $p \in \mathcal{PV}$, we order the elements of X_p according to increasing values of their indices and we build the sequence $\overline{X_p} = (a_1, a_2, \ldots, a_{|X_p|})$, where $a_i \in X_p$. Next, we add the following constraints to our constraint set: $\{(a_i = a_{i+1}) \mid \text{for } a_i \in X_p \text{ and } i = 1, \ldots, (|X_p| - 1)\}$, that is, we require the neighbouring elements of the sequence to be equal. Moreover, for every pair of non-empty sequences $(\overline{P_p}, \overline{N_p})$, where $\overline{P_p} = (a_1, \ldots, a_{|P_p|})$ and $\overline{N_p} = (b_1, \ldots, b_{|N_p|})$, we add a single constraint: $(a_1 \neq b_1)$.

Finally, we take the constant objective function (e.g. $Q(S) = 1$, for $S \in \mathbb{P}$). Then, CPP has a solution iff φ is satisfiable.