SMT versus Genetic and OpenOpt Algorithms: Concrete Planning in the PlanICS Framework

Artur Niewiadomski, Jaroslaw Skaruz
Institute of Computer Science, Siedlce University of Natural Sciences and Humanities
3-Maja 54, 08-110 Siedlce, Poland
artur.niewiadomski@uph.edu.pl; jaroslaw.skaruz@uph.edu.pl

Wojciech Penczek†, Maciej Szreter
Institute of Computer Science, Polish Academy of Sciences
Jana Kazimierza 5, 01-248 Warsaw, Poland
penczek@ipipan.waw.pl; mszreter@ipipan.waw.pl

Mariusz Jarocki
Faculty of Mathematics and Computer Science, Łódź University
Banacha 22, 90-238 Łódź, Poland
jarocki@math.uni.lodz.pl

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

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

*This work has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.
†Also affiliated with: Institute of Computer Science, Siedlce University of Natural Sciences and Humanities
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 [1, 2, 15]. In this paper, we follow the approach of our system PlanICS [5, 6], which has been inspired by [1]. As the main focus of attention is on the methods of service compositions, PlanICS has been designed to be compliant with major web service implementation standards. The semantics represented by OWL ontologies can be processed by commonly used editors and reasoners, so it is easy to adapt other tools and existing semantic data. In turn, WS-BPEL is a natural formalism for exporting the composed plans, so one can use external execution engines for them. Finally, existing WSDL/SOAP services can be easily adapted to the system, with a minimal implementation overhead. The PlanICS system can call web services in a natural, functional manner and a conversion to a state-based model, used internally in PlanICS, can be specified during a registration process, usually without introducing any changes to the web service interface.

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 [12, 16]. Next, offers are retrieved by the offer collector (OC) (a module of PlanICS) and used 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 (shown to be NP-hard) realised by an SMT-planner and compared to GA-based and OpenOpt toolset planners. Fig.1 shows the general PlanICS architecture.

Figure 1. A diagram of the PlanICS system architecture. The bold arrows correspond to computation of a plan, the thin arrows model the planner infrastructure, while the dotted arrows represent the user interactions.
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 other algorithms like GA or openOpt. 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. Our third contribution consists in adapting the numerical computation toolset OpenOpt and showing that it can be used for CPP. However, its effectiveness is in general worse, compared to the other methods described in the paper.

In the last few years the CPP has been extensively studied in the literature. To this aim timed automata and a SAT-based method is used in [14]. The authors of [4] exploit a simple GA to obtain a good quality concrete plan. In [17] 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 [11] 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 three approaches to solving CPP: one based on SMT-solvers, one applying genetic algorithms, and one exploiting the OpenOpt numerical toolset. Sect. 4 discusses experimental results, while the last section summarizes the paper.

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 [13]. A class consists of a unique name and a set of the attributes. By an object we mean an instance of a class.

A key notion of PlanICS is that of a service. We assume that each service processes a set of objects, possibly changing values of their attributes, and produces a set of new (additional) objects. The types of services available for planning are defined as elements of the branch of classes rooted at Service concept. Each service type stands for a description of a set of real-world services of some features. The common features of the services of a given type are described using the attributes which are introduced by the Service class, the ancestor of all the remaining services. These attributes specify three sets of objects, which a service of a given type requires in order to be executed and: (1) leaves them unmodified (the in set), (2) can modify them (the inout set), and (3) produces them (the out set). A service also specifies the
conditions these objects have to satisfy the (pre- and post-Condition), and, what is very important in the context of this paper, describe the interaction between PlanICS and a real-world service of this type.

![PlanICS service model](image)

Figure 2. PlanICS service model. The boxes correspond to in, out, inout of a service, the puzzle- shapes model objects, the dots within them - their attributes.

An intuition behind the service description is presented in Fig.2. The interaction consists in sending to the real-world service input data (“inquiry”), computed by a PlanICS world of objects and receiving from the service output data (“offer”) which affect the new, processed world. This process is performed for these registered services which match requirements originated from a currently composed plan. A large number of services entails a significant amount of possible interactions. This implies, in turn, a large number of offers, which justifies the stage of a concrete planning, i.e., matching and optimizing offers gathered in the previous stage.

Below, we present a simplified ontology, used further as a running example.

**Example 2.1. (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 2.2. (World, Abstract World)**
Let \( D = \mathbb{Z} \cup \mathbb{R} \cup A \), where \( \mathbb{Z} \) is the set of integers, \( \mathbb{R} \) is the set of real numbers, and \( A = \{ \text{set}, \text{null} \} \) is the set of abstract values. Let \( \mathcal{O} \) be the set of all objects, \( A \) be the set of all attributes, and \( \text{attr} : \mathcal{O} \rightarrow 2^A \) be the function returning the attributes of an object. A world is a pair \((\mathcal{O}, \text{val})\), where \( \mathcal{O} \subseteq \mathcal{O} \) is a set of objects and \( \text{val} : \mathcal{O} \times \text{attr}(\mathcal{O}) \rightarrow D \) is a valuation function, which to every attribute of the objects from \( \mathcal{O} \) 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 \( A \).
Since the main part of this paper deals with an optimization problem, the domains under consideration are the integer and the real numbers\(^1\). 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 \(s\) [13]. 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 \(w \xrightarrow{s,cx} w'\) [12].

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. 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 2.3. (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 \(type \in \{\text{Sum}, \text{Min}, \text{Max}\}\). 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

\(^1\)Note that other types of values used in PlanICS framework, like strings, dates, and Boolean values can be easily coded by integers.
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 ∼, and as a result we get a boolean value.

**Example 2.4. (Query specification)**

Consider the ontology from Example 2.1. 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: \( \text{IN} = \emptyset, \text{IO} = \{ m : \text{Money} \}, \text{OU} = \{ t_1, t_2, t_3 : \text{Transaction} \}, \text{Pre} = (m.\text{amount} \leq 100), \text{and} \text{Post} = (t_1.\text{amount} + t_2.\text{amount} > 50) \). 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:

\[
\text{Aggr} = \{(\text{Charge}, \text{true}, \text{fee}, \text{Max}, <, 3)\}, \text{and} \text{Qual} = \{(\text{Transaction}, \text{true}, \text{profit}, \text{Sum})\}.
\]

Formally, a user query is defined as follows:

**Definition 2.5. (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 [12], using the following definition.

**Definition 2.6. (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 \(a \in \text{attr}(o)\) \(val'(o, a) = \begin{cases} \text{set, for } val(o, a) \neq \text{null}, \\ \text{null, for } val(o, a) = \text{null}. \end{cases}\)

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

**Definition 2.7. (Transformation sequence)**

Let \(k\) be a natural number and \(seq = ((s_1, cx_1), \ldots, (s_k, cx_k))\) 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 \((w_1, w_2, \ldots, w_k)\) such that \(\forall 1 \leq i \leq k\) \(w_i \leftarrow s_i, cx_i, w_{i-1}\). 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 2.8. (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 [12, 13].

**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 2.9. (Offer, Offer set)**
Assume that the $n$-th instance of a service type from a CAP (the $n$-th CAP node) processes some number of objects having in total $m$ attributes\(^2\). 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 2.10. (Offer, Offer sets)**
Consider the user query from Example 2.4 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.

---

\(^2\)Note that we consider two copies of the inout objects: taking values from before and after modification by the service.
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 maximizing a quality function. Thus, we can formulate CPP as a constrained optimization problem.

**Definition 2.11. (CPP)**

Let \( n \) be the length of CAP and let \( \Omega = (O^1, \ldots, O^n) \) 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 \( (P^i_1, \ldots, P^i_n) \), such that \( j_i \in \{1, \ldots, k_i\} \) and \( i \in \{1, \ldots, n\} \). The Concrete Planning Problem is defined as:

\[
\max \{ Q(S) \mid S \in \mathbb{P} \} \text{ subject to } C(S),
\]

where \( Q : \mathbb{P} \mapsto \mathbb{R} \) is an objective function defined as the sum of all quality constraints and \( C(S) = \{ C_j(S) \mid j = 1, \ldots, c \text{ for } c \in \mathbb{N} \} \), 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 2.12.** The concrete planning problem (CPP) is NP-hard.

**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 \( PV \) and 3-CNF formula \( \varphi = c_1 \land c_2 \land \cdots \land c_n \), where \( c_i = (x^1_i \lor x^2_i \lor x^3_i), x^1_i = p \) or \( x^1_i = \neg p \), for every \( p \in PV, i = 1 \ldots n \) and \( j = 1 \ldots 3 \). We encode the satisfiability problem of \( \varphi \) 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 \( 8 \times 3 \) matrix. Thus, \( \mathbb{P} \) is the set of all possible binary sequences of length \( 3n \).

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

The above lemma gives the lower bound on the time complexity of CPP.

Consider now the upper bound under the assumption that the quality function takes a finite number of (known) values and the complexity of checking all the constraints of a sequence of offers is polynomial. Let \(N\) be the number of the values of the quality function. Our algorithm \(N\) times non-deterministically generates a sequence of offers by selecting one offer from each offer set and only then checks whether all the constraints are satisfied and the value of the objective function is equal to a selected one. The values of the objective function are selected one by one in the decreasing order starting from the maximal one. This shows that CPP, under the above assumption, is in NP with respect to the size of a plan. Without any assumption on the objective function it is easy to show that the upper bound of CPP is in EXPTime. The deterministic algorithm generates all the sequences of offers by selecting one offer from each offer set and for each sequence checks whether all the constraints are satisfied. Then, the sequences satisfying all the constraints and having the maximal value of the objective function are selected.

3. Approaches to Concrete Planning

In this section we present three approaches to the concrete planning problem.

3.1. Concrete Planning using SMT

First, we describe a 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 \(V\) of all necessary SMT-variables (for simplicity called just variables) is allocated:

- \(q\) - for storing the subsequent values of the quality function found,
- \(oid^i\), 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 \(oid^i\) variable takes a value ranging over \(1, \ldots, k_i\), where \(k_i\) corresponds to the number of the offers collected from the services matching the \(i\)-th CAP node, i.e., the number of the rows in the \(O^i\) matrix.
- \(o^i_j\), where \(i = 1 \ldots n\), \(j = 1 \ldots m_i\), and \(m_i\) is the number of the values constituting an offer in the \(i\)-th offer set, that is, the number of the columns in the \(O^i\) matrix. 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 \(V\), the offer values from the offer sets \(\emptyset = (O^1, \ldots, O^n)\) are encoded as the formula

\[
ofr(\emptyset, V) = \bigwedge_{i=1}^{n} \bigwedge_{d=1}^{k_i} \left( oid^i = d \land \bigwedge_{o^i_{d,j} \in R^i} o^i_j = o^i_{d,j} \right).
\]  

(2)
Then, the conjunction of all constraints is encoded as the formula $ctr(C(S), V)$, and the objective function as the formula $qual(Q(S), V)$. For convenience its value is bound with the variable $q$ by $q = qual(Q(S), V)$. Thus, the formula encoding the solutions of CPP is as follows:

$$cpp(\emptyset, Q(S), C(S), V) = ofr(\emptyset, V) \land ctr(C(S), V) \land q = qual(Q(S), V) \quad (3)$$

The algorithm of searching for the maximal value of $q$ is an adaptation of the binary search method exploiting the assumptions mechanism of an SMT-solver. This mechanism consists in checking satisfiability of an SMT-formula assuming that a set of boolean conditions is satisfied. In order to set the initial search interval, the minimal and the maximal values of the objective function are computed. In every iteration the searched interval is divided in half and, since the objective function is to be maximized, a solution of value greater than a half ($pivot = (\text{min} + \text{max})/2$) is searched. To this end the whole formula is checked for satisfiability under the assumption ($q > pivot$). If there is a solution, then its value becomes $\text{min}$. Otherwise, the searched value is less or equal $pivot$, the last assumption is replaced by its negation, and $pivot$ value is assigned to $\text{max}$. Then, a new $pivot$ value is computed and the algorithm is iterated again, until the difference between $\text{max}$ and $pivot$ becomes smaller than the given parameter $delta$, which enables the search with the given accuracy.

### 3.2. Concrete planning using Genetic Algorithms

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 a standard GA. We treat our implementation of GA as a benchmark for the SMT-based planner. The objective of GA is to find a concrete plan such that it satisfies all the constraints and the obtained optimization objective is maximal. At each generation of GA, the individuals are evaluated by calculating their fitness values which are later used by the selection operator. It selects individuals from the current population to the temporary one with the probability proportional to their fitness values. Next, the obtained individuals take part in genetic modifications through the standard crossover and mutation operators.

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. 2.11), multiplied by some constant $\beta$:

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

where $Ind$ stands for an individual, $S_{Ind}$ is a sequence of the offer values corresponding to $Ind$, $sat(C(S_{Ind}))$ 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 $\beta$ is to reduce both components of the sum to the same order of magnitude and to control the impact of the components on the final result.

### 3.3. Concrete planning using the optimization framework OpenOpt

The formulation of the concrete planning problem as an optimization over discrete domains raises a question whether a state-of-art numerical optimization tool could be applied and compared to our approaches. There are several such tools, with [9, 10] being particularly popular, but besides licence issues there is a problem of handling discrete domains. Most of the tools for solving equation systems do not support
the most flexible with respect to specification capabilities. It accepts Python programs as an input, which enables mixing Boolean conditions and arithmetic constraints, thus making it possible to express our optimization task. OpenOpt can use several numerical solvers, both freely available and commercial ones, but we have found only its own 'interalg' solver able to deal with our discrete domains of tuple sets.

The encoding in Python was rather straightforward for variable declarations, constraints, quality function, and variable domains. A more tricky part was the case when more than one variable from a service was relevant, as we needed to encode that all the variable values belong to the same tuple.

We have noticed that while it is possible to encode the problem, the solver cannot deal with more than the simplest cases. In order to make a search easier we have examined pruning of the searched space at the price of losing the guarantee of finding the optimal solution. First, the solver is started to find the optimal solution, with constraints present but with the domain restricted only by the variable ranges. This problem is solved very quickly. Then, only the offers being close to the global solution are chosen to the detailed search and the solver is run again.

One might argue that in the presented area, for real-time applications it can sometimes be desirable to prefer speed over accuracy if it results in finding reasonably good solutions.

4. Experimental Results

In this section we compare the results of several experiments obtained applying our three planning methods. The experiments have been performed using the Z3 SMT-solver (version 4.3) running on a standard PC equipped with 2.8 GHz CPU. In each of the experiments we use different optimization objectives and constraints. Equation 5 presents the objectives $Q_1, Q_2, Q_3$ used in the experiments 1-5, while the constraints are combinations of $C_1, C_2, C_3$ defined by Equation 6. In the experiments 6 and 7, we use the constraints and the objective function of our working example.

$$
Q_1 = \sum_{i=1}^{n} a_{ji,1}^i, \quad Q_2 = \sum_{i=1}^{n} a_{ji,2}^i, \quad Q_3 = \sum_{i=1}^{n} (a_{ji,1}^i + a_{ji,2}^i), \quad (5)
$$

$$
C_1 = \{(a_{ji,1}^i < a_{ji+1,1}^i)\}, \quad C_2 = \{(a_{ji,2}^i < a_{ji+1,2}^i)\}, \quad C_3 = \{(a_{ji,2}^i = a_{ji+1,2}^i)\} \quad \text{for } i = 1, \ldots, n-1. \quad (6)
$$

The offer sets used in all the experiments have been randomly generated by our Offer Generator. 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- and OpenOpt-based algorithms, the 500 sec. time-out has been set. Each experiment has been repeated 20 times as the run-times obtained have been very similar, and the quality values have been the same each time. In all experiments we have tested instances with 5, 10, and 15 offer sets, containing 256, 512, and 1024 offers each.

In Experiment 1 $Q_1$ was used as an optimization objective and $C_1$ as a set of constraints. The results are presented in Table 1 (left). The meaning of the columns is the following: Sp. - the search space size, $n$ - the plan length, $Offs$ - the number of offers in each set, $t[sec]$ - the time consumed by the planners, $Q$
and $AvgQ$ - the quality of plans found, $Bs$ and $Pr.$ - the probability of finding an optimal solution and of finding any solution by GA. The general observations are very encouraging. All the solutions have been found quickly in several seconds only. The SMT-based planner, as well as the OpenOpt one, always return optimum, while GA, as a non-deterministic algorithm, finds optimum in at most 16\% cases, for instances with 5 service types. Moreover, we can observe that for the plans of length 15 the results of GA are much worse than for the shorter plans. In this case the probability of finding a solution drops below 20\%, and the average quality of solutions differs from the optimum up to about 26\%.

Table 1. The results of Experiment 1 (left) and Experiment 2 (right).

<table>
<thead>
<tr>
<th>Sp.</th>
<th>n</th>
<th>Offs</th>
<th>SMT</th>
<th>GA</th>
<th>oOpt</th>
<th>SMT</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>t[s]</td>
<td>Q</td>
<td>AvgQ</td>
<td>Bs</td>
<td>Pr.</td>
</tr>
<tr>
<td>2^{45}</td>
<td>5</td>
<td>512</td>
<td>0.29</td>
<td>485</td>
<td>1.53</td>
<td>478.51</td>
<td>16</td>
</tr>
<tr>
<td>2^{40}</td>
<td>1024</td>
<td>1.00</td>
<td>490</td>
<td>1.99</td>
<td>479.98</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>2^{30}</td>
<td>256</td>
<td>0.43</td>
<td>920</td>
<td>2.99</td>
<td>741.85</td>
<td>85</td>
<td>5.2</td>
</tr>
<tr>
<td>2^{30}</td>
<td>512</td>
<td>0.79</td>
<td>955</td>
<td>3.43</td>
<td>775.57</td>
<td>93</td>
<td>0.7</td>
</tr>
<tr>
<td>2^{50}</td>
<td>1024</td>
<td>1.00</td>
<td>955</td>
<td>4.29</td>
<td>805.68</td>
<td>95</td>
<td>0.49</td>
</tr>
<tr>
<td>2^{120}</td>
<td>256</td>
<td>0.64</td>
<td>1350</td>
<td>5.02</td>
<td>993.62</td>
<td>8</td>
<td>1.32</td>
</tr>
<tr>
<td>2^{135}</td>
<td>512</td>
<td>1.30</td>
<td>1377</td>
<td>5.59</td>
<td>1018.8</td>
<td>15</td>
<td>0.91</td>
</tr>
<tr>
<td>2^{130}</td>
<td>1024</td>
<td>1.85</td>
<td>1395</td>
<td>7.09</td>
<td>1057.7</td>
<td>19</td>
<td>1.53</td>
</tr>
</tbody>
</table>

In Experiment 2 the constraints remain the same ($C_1$), the objective function is similar ($Q_2$), but we maximize the sum of other values than these used in the constraints. This entails increasing the number of relevant variables used to encoding the problem by SMT- and OpenOpt-based planners. Unfortunately, OpenOpt is unable to find any solution within the given time limit, while working on the whole search space. Thus, Table 4 presents additional results obtained after pruning the state space.

Table 1 (right) presents the results of Exp. 2 obtained by SMT and GA. 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.

In order to discover the limitations of both the planners, 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 $C_1$ as the set of constraints. Moreover, we aim at examining the behaviour of the SMT-solver in the presence of a larger number of constraints. Thus, in Exp. 4 we use $Q_3$ as the optimization objective and $C_1 \cup C_2$ as the set of constraints. Table 2 presents the results. While these instances are in general harder to solve for SMT, several times the given time limit has been reached. In these cases we put TO instead of the solver run-time in the tables, and the obtained quality values are marked with the asterisk, while we have no guarantee that they are optimal ones.

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 6\% up to about 18\%, 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.
Table 2. The results of Experiment 3 (left) and Experiment 4 (right).

<table>
<thead>
<tr>
<th>Sp.</th>
<th>n</th>
<th>Offs</th>
<th>SMT</th>
<th>GA</th>
<th>SMT</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>t[s]</td>
<td>Q</td>
<td>t[s]</td>
<td>AvQ</td>
</tr>
<tr>
<td>(2^{15})</td>
<td>5</td>
<td>512</td>
<td>3.75</td>
<td>924</td>
<td>1.88</td>
<td>869.21</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>10</td>
<td>1024</td>
<td>2.55</td>
<td>947</td>
<td>2.31</td>
<td>893.1</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>10</td>
<td>256</td>
<td>TO</td>
<td>1547*</td>
<td>3.61</td>
<td>1445.88</td>
</tr>
<tr>
<td>(2^{10})</td>
<td>10</td>
<td>512</td>
<td>306</td>
<td>1803</td>
<td>4.11</td>
<td>1490.07</td>
</tr>
<tr>
<td>(2^{10})</td>
<td>10</td>
<td>1024</td>
<td>213</td>
<td>1862</td>
<td>4.98</td>
<td>1532.46</td>
</tr>
<tr>
<td>(2^{123})</td>
<td>15</td>
<td>256</td>
<td>TO</td>
<td>2266*</td>
<td>5.95</td>
<td>2085.4</td>
</tr>
<tr>
<td>(2^{123})</td>
<td>15</td>
<td>512</td>
<td>2409*</td>
<td>6.64</td>
<td>2001.85</td>
<td>7</td>
</tr>
<tr>
<td>(2^{123})</td>
<td>15</td>
<td>1024</td>
<td>2637*</td>
<td>8.2</td>
<td>2194.53</td>
<td>13</td>
</tr>
</tbody>
</table>

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, the quality of which differ from optimum by 13% to about 33%. Secondly, adding more constraints improves slightly the efficiency of the SMT-based planner. 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 \(C_1 \cup C_3\) 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 3. The SMT-runtime decreases tremendously, as well as the quality and the probability of finding a solution by GA. Now, GA barely finds solutions of length 5, the quality of which differ from optimum even by 45%.

Experiments 6 and 7 are based on our working example. Table 3 presents the results, where NO means that there is no solution at all. In Exp. 6 we used 6, 11, and 16 constraints for plans of length 5, 10, and 15, respectively. The quality of solutions found by GA is worse by about 26% to 55% than the ones found by the SMT-solver. Moreover, besides the shortest plans, 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 31 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. In this case GA finds solutions for only one instance with 5 services and 1024 offers, with probability equal to 22% and average quality equal to 149.18 (not shown in the table).

Table 4 shows the results for Exp. 2 (left) and Exp. 4 (right), with the state spaces pruned so that only offers with the relevant variables greater than the threshold \(P\) are considered. It can be seen that too small threshold values result in longer execution times (possibly exceeding the limit), while too big values result in finding suboptimal solutions or not finding them at all. All these cases are shown in Table 4, in the rows concerning the plans of length 10 and 512 offers. The other table rows show only the best prune threshold.

For example, in Exp. 2, after pruning the offer sets with threshold 90 OpenOpt finds the optimal solution, but it consumes over 400 sec. It should be mentioned that the size of search space is about
2^{48}$ in this case. After more thorough pruning with threshold 97 we obtain the search space of size $2^{31}$. Now OpenOpt is able to find optimum in about 2 sec. However, if the threshold increases, then in the remaining search space (of size $2^{24}$) there is no valid solution.

## 5. Conclusions

In the paper we have presented the concrete planning in the Plan!cs framework, its time complexity, and its reduction to the constrained optimization problem together with a new SMT-based approach to solve it. The experimental results, compared with results of the standard GA and the OpenOpt toolset, present advantages and disadvantages of 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 can find 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. Concerning OpenOpt, we have shown that it can also be used for solving CPP. However, its effectiveness is satisfactory only if no tuples of values are present in the problem domain and much worse in the opposite case. The toolset does not seem to be optimized for this type of discrete domains, and this forms an interesting research topic in the numerical optimization, possibly combining pure numeric methods with combinatorial ones.

We could only make OpenOpt working by pruning search spaces, but, it is important to mention that this can be done for specific quality functions and variable domains only.

References


