Towards Automatic Composition of Web Services: SAT-Based Concretisation of Abstract Scenarios*

Artur Niewiadomski  
Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, Poland

Wojciech Penczek†  
Institute of Computer Science, PAS, Warsaw, Poland  
penczek@ipipan.waw.pl  
and Siedlce University of Natural Sciences and Humanities, Poland

Agata Póroła  
Faculty of Mathematics and Computer Science, University of Łódź, Poland

Maciej Szreter  
Institute of Computer Science, PAS, Warsaw, Poland

Andrzej Zbrzezny  
Institute of Mathematics and Computer Science, Jan Długosz University in Częstochowa, Poland

Abstract. Automating the composition of web services is an object of a growing interest. In our paper [13] we proposed a method for converting the problem of the composition to the problem of building a graph of worlds consisting of formally defined objects, and presented the first phase of this composition aimed at building a graph of types of services (an abstract graph). In this work we propose a method of replacing abstract flows of this graph by sequences of concrete services able to satisfy the user’s request. The method is based on SAT-based reachability checking for (timed) automata with discrete data and parametric assignments.

*Partly supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.
†Address for correspondence: Institute of Computer Science, PAS, Jana Kazimierza 5, 01-248 Warsaw, Poland
1. Introduction

In recent years there has been a growing interest in automating the composition of web services. The number of complex services is rapidly growing nowadays, but still there is a lack of automatic methods for arranging and executing their flows. One of the problems to deal with is the size of the environment: most existing composition methods work with concrete instances of web services, so even a simple query requires taking all the instances of all the types of services into account. Another problem follows from incompatibilities in inputs/outputs of services, and difficulties in comparing their capabilities and qualities - two services can offer the same functionality, but this cannot be detected automatically without unification of their interfaces made by the providers.

In our work [13] we proposed an approach to automatic composition of services which can potentially solve the above problems. The problem of composition is converted to the problem of building a graph of worlds consisting of formally defined objects which are transformed by services. We introduce a uniform semantic description of service types. In order to adapt a possibly wide class of existing services, including stateful, as well as stateless ones, specific interfaces of concrete services are to be translated to the common one by adapters (called proxies), built in the process of service registration. The process is to be based on descriptions of interfaces of services, specified both in WSDL and in a language containing a semantic information (like OWL-S or Entish [1]). The client’s goal is expressed in a fully declarative intention language. The user describes two worlds: the initial and the final one, using the notions coming from an ontology, and not knowing any relations between them or between the services. The task of the composition system consists in finding a way of transforming the initial world into the final one. The composition is three-phase. In the first phase, called abstract planning or planning in types, we create an abstract plan, which shows sequences of service types whose executions possibly allow to accomplish the goal. The second phase makes these scenarios “concrete”, which means replacing the types of services by their concrete instances. Finally, the last phase consists in supervising the execution of the optimal run, with a possibility of correcting it in the case of a service failure.

Our previous paper [13] described a method of generating an abstract graph of services. The current work deals with the second phase of composition: concretising abstract flows, i.e., searching for sequences of concrete services which can lead to satisfying user’s request. We apply model checking techniques to this aim. The sub-stage aimed at choosing an optimal scenario is not considered in this version of the approach.

The rest of the paper is organised as follows. In Sec. 2 we present the related work. Sec. 3 introduces worlds and services transforming them. Sec. 4 describes briefly the abstract planning phase and its result. Sec. 5 presents our approach to SAT-based concretising abstract scenarios. Sec. 6 and Sec. 7 show experimental results and concluding remarks.

2. Related Work

There exist many papers dealing with the topic of web services composition (WSC) [17, 21, 22, 23, 24, 28], presenting either complete systems or partial solutions usable in WSC. Some of them consider static approaches where flows are given as a part of the input, while the others deal with dynamically created flows [23]. One of the most active research areas is a group of methods referred to as AI Planning [17, 12]. Several approaches use Planning Domain Definition Language (PDDL [18]). Another group of methods
is built around the so-called rule-based planning, where composite services are generated from high-level declarative descriptions, and compositionality rules describe the conditions under which two services are composable. The information obtained is then processed by some designated tools. The project SWORD [20] uses an entity-relation formalism to specify web services. The services are specified using pre- and postconditions; a service is represented as a Horn rule denoting that the postcondition is achieved when the preconditions are true. A rule-based expert system generates a plan. In [29] Vitvar et al. proposed a solution based on the WSMO/WSML [25] formalisms.

Another related methodology is the logic-based program synthesis [22]. Besides the automatic approaches mentioned above, there exist also half-automatic methods assuming human assistance at some stages [27].

The most complete specification of an automatic composition system was described in [1]. It is based on a multi-phased composition using a uniform semantic description of services. Besides of complete systems’ descriptions there is also a number of various methods involved in solving the WSC problem. The simplest ones are based on explicit state space search algorithms [26], while the more advanced ones employ, among others, graph-based planning [4, 5], logic programming [20], AI planning [17], genetic algorithms [2] and model checking methods [11, 16]. The latter involve mainly symbolic techniques: in the case of MIPS [9], BDDPLAN [11], and UMOP [14] the planning methods are based on Ordered Binary Decision Diagrams [6], while another group of approaches makes use of methods solving the Boolean satisfiability problem (SAT) [3]. Planning procedures based on these techniques are for example SATPLAN [15], BLACKBOX [16], and LGP [10].

Inspired by the Entish project [1], our approach to automated composition is based on matching input and output types of services. We adapt also the idea of three-phase composition, but introduce original definitions of services and composition techniques. The concrete planning presented in this work (used in the second phase of the composition process) makes use of SAT-based bounded model checking method applied to testing reachability for an extended timed automaton obtained for an abstract plan resulting from the first phase of the composition, which is our original solution.

3. Worlds and Services

In our approach we introduce a unified semantics for functionalities offered by services, which is done by defining a dictionary of notions/types describing their inputs and outputs. A service is understood as a function which transforms a set of data into another set of data. The sets of data are called worlds. The worlds can be described by the use of an ontology, i.e., a formal representation of the knowledge about them. Formally, an ontology is a set of definitions of classes (ordered in an inheritance hierarchy), their instances (objects), and relations between them.

Due to the lack of space we provide only a short description of the above notions (a detailed formalisation can be found in [13]). So, a world is a set of objects, each of which contains named attributes. An object can be a concrete or an abstract one; attributes of concrete objects can be values of simple types (numbers, strings, boolean values etc.) or NULL (an empty value), whereas these of abstract objects carry an information whether the value of the attribute is set, not set, or “set or unset” (which is expressed by the values NULL, SET, and ANY, respectively). An attribute of an object can be also an object itself. Moreover, each attribute has a boolean-valued flag const, which specifies whether modifying the value of this attribute is allowed. The attributes are referred to by ObjectName.AttributeName. In order to
reason about worlds we can use the functions Exists, isSet, and isConst. Intuitively, Exists specifies whether an object exists in a world, isSet says whether an attribute of an object is set, and isConst informs whether the value of an attribute of an object can be changed.

**Example 1.** Consider the following simple ontology:

```plaintext
class Thing
class Ware extends Thing [id : integer; name, owner : string]
class Measurable extends Thing [capacity : float]
class Juice extends Ware, Measurable
class Fruits extends Ware, Measurable
```

In the brackets there are the lists of attributes (together with their types). Notice that the classes Juice and Fruits contain all the attributes of Ware and Measurable.

The ontologies collect the knowledge not only about the structure of worlds, but also about the ways they can be transformed, i.e., about services. The services are organised in a hierarchy of classes, and described both on the level of classes (by specifying what all the services of a given class do - such a pattern of behaviour is referred to as an **abstract service**), and on the level of objects (**concrete services**). A service is defined to be an object of a non-abstract subclass of the abstract class Service. A service contains (initialised) attributes which can be grouped into **processing lists** (the attributes produces, consumes, requires), **modification lists** (the attributes mustSet, maySet, mustSetConst, maySetConst), and **validation formulas** (the attributes preCondition and postCondition).

A service modifies a world, which involves both modifying the set of objects present, and the values of their attributes. Intuitively, the attribute consumes of a service is a list of objects which "disappear" from a world when the service is invoked, produces lists the objects which are created by the service, while requires is a list of objects whose existence is not influenced by the service, but the values of their attributes can be changed. The attributes which are (or can potentially be) modified by the service are listed in mustSet and maySet; similarly, mustSetConst and maySetConst specify the attributes whose flags const are (or can potentially be) set true. The formula preCondition gives the conditions which should be satisfied by the world at which the service is to be executed (its **input world**), whereas postCondition specifies the conditions which are satisfied by the world obtained after the service is executed (**output world**).

**Example 2.** Consider the following services extending the ontology shown in Example 1:

```plaintext
class Selling extends Service: [
    produces = null; consumes = null; requires = w:Ware; mustSet = w.id, w.owner;
    preCondition = not isSet(w.id) and isSet(w.name) and isSet(w.owner);
    postCondition = w.owner!=pre(w).owner and w.id>0]
class FruitSelling extends Selling: [
    requires = w:Fruits; mustSet = w.capacity; postCondition = w.capacity>0 ]
class JuiceSelling extends Selling: [
    requires = w:Juice; mustSet = w.capacity; postCondition = w.capacity>0 ]
```

---

We use the standard terminology of object-oriented programming. The term "subclass" is related to inheritance. A class is called abstract if instantiating it (i.e., creating objects following the class definition) is useless, in the sense that the objects obtained this way do not correspond to any real-world entity.
The description of the class Selling means that all the services of the type Selling can affect a world which contains an object of the class Ware which is given a name and an owner, but no id. The service sets the id and the owner of the ware. Additionally, the postcondition implies that the new owner differs from the previous one. The class FruitSelling (extending Selling) limits the worlds which can be affected to these containing an object of the class Fruits. Besides setting id and the owner, it additionally sets the capacity, and extends the postcondition of Selling by requiring a positive value of this capacity. The descriptions of the rest of classes can be interpreted in a similar way.

3.1. Concrete Services

In this work we assume that concrete services present their offers in their pre- and postconditions. and that their maySetConst and maySet lists are empty (i.e., setting an attribute or the const flag optionally is not allowed). A grammar for validation formulas is as follows:

\[
\begin{align*}
\text{ObjectName} & ::= \text{ObjectName from consumes} | \text{ObjectName from produces} | \text{ObjectName from requires} | \text{ObjectName from requires} \\
\text{ObjectAttribute} & ::= \text{ObjectName}.\text{AttributeName} | \text{ObjectName}.\text{ObjectAttribute} \\
\text{ExpressionElement} & ::= \text{"integer value"} | \text{"real value"} | \text{ObjectAttribute} \text{"of a numeric type"} \\
\text{ArithmOp} & ::= + | - | * | / \\
\text{Expression} & ::= \text{ExpressionElement} | \text{Expression} \text{ArithmOp} \text{Expression} \\
\text{CompOp} & ::= = | < | <= | > | >= \\
\text{AtomicPredicate} & ::= \text{Exists(ObjectName)} | \text{isSet(ObjectAttribute)} | \text{isConst(ObjectAttribute)} | \text{not AtomicPredicate} | \text{ObjectAttribute} \text{CompOp} \text{Expression} \\
\text{ObjectAttribute} \text{CompOp} \text{"value"} \\
\text{Conjunction} & ::= \text{AtomicPredicate} | \text{AtomicPredicate and Conjunction} \\
\text{ValidationFormula} & ::= \text{Conjunction} | \text{Conjunction or ValidationFormula}
\end{align*}
\]

It should be noticed that pre() and post() are allowed in postCondition only. Moreover, we assume that the expressions involve only attributes which refer either to the names of objects from consumes, or to the names of objects which are of the form pre(objectName from requires) (i.e., assume that the expressions involve only values of attributes from the input world of a service). We assume also that all the elements of an expression are of the same type, the result is of this type as well, and so is the attribute this result is compared with. The expressions involve attributes of numeric types only, while the atomic predicates allow comparing an attribute of an arbitrary type with a value of the same type\(^2\).

The values of the attributes and the values occurring in comparisons in the atomic predicates above are: boolean values, integer values of a certain range\(^3\), characters, real values of a certain range with the precision limited to a number of decimal places, and values of certain enumeration types. Enumeration types are used instead of strings. Such an approach seems sufficient to represent the values necessary: in most cases the names of items offered or processed by services come from a certain set of known names.

\(^2\)The grammar for the validation formulas is given in a semi-formal way. The “quoted” items should be understood as notions from the natural language. By “value” we mean a value of an arbitrary (also non-numeric) type.

\(^3\)A natural restriction when using programming languages.
(e.g. names of countries, cities, washing machines types etc), or can be derived from the repository (e.g. names of shops which registered their offers). Similarly, restricting the precision of real values seems reasonable (usually two decimal places are sufficient to express the amount of a ware we buy, a price, a capacity etc). Consequently, all the values considered can be treated as discrete. It should be noticed also that we assume an ordering on the elements of enumeration types and the boolean values.

Example 3. Assume that we have the following types: integer, float (we can assume that the precision is up to two decimal places), FruitTypes = (strawberry, blueberry, apple, plum), and OwnerNames = (Me, Shop1, Shop2, Shop3, Shop4). The concrete instances of the service classes listed in Example 2 can be as follows:

object FruitNetMarket instanceOf SelectWare: 
  postCondition = (w.name=strawberry and w.owner=shop1) or (w.name=blueberry and w.owner=shop1)

object FruitNetOffers instanceOf SelectWare: 
  postCondition = (w.name=plum and w.owner=shop2) 
  or (w.name=apple and w.owner=shop2) 
  or (w.name=apple and w.owner=shop3)

object Shop1 instanceOf FruitSelling: 
  precondition = w.owner=shop1; postCondition = w.capacity<=10

object Shop2 instanceOf FruitSelling: 
  precondition = w.owner=shop2

object Shop3 instanceOf FruitSelling: 
  precondition = w.owner=shop3; postcondition = w.capacity>=100

object HomeJuiceMaking instanceOf MakingJuice: 
  preCondition = f.capacity<=10 and f.name!=plum and f.name!=apple; 
  postCondition = j.name=f.name and j.capacity=f.capacity and j.owner=f.owner

object GrandmaKitchen instanceOf MakingJuice: 
  preCondition = f.capacity<=5; 
  postCondition = j.name=f.name and j.capacity=f.capacity and j.owner=f.owner

object JuiceTex instanceOf MakingJuice: 
  postCondition = j.name=f.name and j.capacity=2*f.capacity and j.owner=f.owner

The selecting service FruitNetMarket offers strawberries or blueberries from Shop1, while FruitNetOffers - plums from Shop2 or apples from Shop2 or Shop3. The (fruit)selling service Shop1 declares to sell only the fruits it owns, limiting their capacity to not more than 10 units. Shop3 can sell not less than 100 units, while Shop2 does not limit the capacity of fruits sold. Similarly, the services which offer making juice from fruits provide some limitations on the amount and the kind of the fruits they take, and vary in the capacity of juice they can produce from a given amount of fruits. Notice that the “effective” pre- and post-conditions of the concrete services are obtained by conjuncting their own formulas with these for the classes (service types) they belong to.

4. Abstract Planning

The aim of the composition process is to find a sequence of services whose execution satisfies a user’s goal. The user describes its goal in a declarative language defined by the ontology. Its query contains (among others, see [13] for a full description) an initial domain, i.e., a list of named objects, which are elements of the initial world; an initial clause specifying a condition, which is to be satisfied by the initial world; an effect domain - a list of named objects which have to be present in a final world; and an effect clause specifying a condition, which is to be satisfied by the final world.

4Similarly as in the Ada programming language.
Example 4. A user’s query referring to the ontology from the previous examples can be of the form

\[
\begin{align*}
\text{InitWorld} &= \text{null}; \\
\text{InitClause} &= \text{true}; \\
\text{EffectWorld} &= \text{j:Juice}; \\
\text{EffectClause} &= \text{j.id}>0 \text{ and j.capacity}=10 \text{ and j.owner} = \text{"Me"}
\end{align*}
\]

which means that the user wants to become the owner of 10 units of juice, starting from the empty initial world.

The aim of the composition process is to find a path in the graph of all the possible transitions between worlds which leads from a given initial world to a given final world, specified (possibly partially) in a user’s query, using no other knowledge than that contained in the ontology. The composition is three-phase: the first phase (described in [13]) consists in finding all the sequences of service types (abstract services) which can potentially lead to satisfying the user’s goal. The result of the abstract planning phase is an abstract graph - a directed multigraph whose nodes are worlds in certain states (i.e., having certain values of the attributes of their objects) while its edges are labelled by services. The nodes with no input edges (initial nodes) represent alternative initial worlds, while these with no output edges are alternative final worlds. A formal definition of the abstract graph can be found in [13].

Example 5. The sequences of types of services whose execution can lead to satisfying the user’s request from Example 4 are (1) SelectWare then FruitSelling and then MakingJuice, (2) SelectWare and then Juice-Selling, (3) SelectWare, then Selling and then MakingJuice, and (4) Selectware and then Selling (see [13]).

5. Main Idea

The phase of abstract composition [13] produces a graph, in which the sequences of service types can potentially satisfy the user’s request. The next phase of the composition process, aimed at obtaining a flow to be run, is to find concrete services of the appropriate types whose offers make the query satisfied. To this aim SAT-based Bounded Model Checking (BMC) method is used. It is shown in [31] how to test reachability for timed automata with discrete data using the BMC module of the model checker Veric. In this paper we adapt the above approach.

The main idea of our solution consists in translating each path of the graph, resulting from the abstract composition phase, to a timed automaton with discrete data and parametric assignments (TADDPA). The automaton represents the concrete services of the appropriate types (corresponding to the types of services in the selected scenario), which can potentially be executed to reach the user’s goal. The variables of the automaton store the values of the attributes of the objects occurring along the path, while the parameters are assigned to variables when the exact value to be assigned by a service is unknown. Next, reachability of a state satisfying the user’s query is checked. If such a state is reachable, then a reachability witness is obtained, containing both the information about a sequence of concrete services to be executed to satisfy the goal and values of the parameters for which this sequence is executable.

In spite of using TADDPA we currently do not make any use of the timing part of this formalism. However, the motivation for using TADDPA is twofold. Firstly, the existing implementation for TADD (modified to handle their extension to TADDPA) can be adapted. Secondly, our next step in developing the composition tool is to use clocks to represent the declared times of services executions, which should enable us searching for scenarios satisfying some time constraints.

Below, we introduce all the formal elements of our approach.
5.1. Timed Automata with Discrete Data and Parametric Assignments

Given a set of discrete types $\mathcal{T} = \bigcup_{i=1}^{n} T_i$ ($n \in \mathbb{N}$), including an integer type, a character type, user-defined enumeration types, a real type of a precision given etc., such that for any $T_i \in \mathcal{T}$ there is an ordering\(^5\) on the values of $T_i$. By $\mathcal{T}_N \subset \mathcal{T}$ we denote the subset of $\mathcal{T}$ containing all the numeric types $T_i \in \mathcal{T}$. Let $DV$ be a finite set of variables whose types belong to $\mathcal{T}$, and let $DP$ be a finite set of parameters whose types belong to $\mathcal{T}$. Let $type(a)$, for $a \in DV \cup DP$, denote the type of $a$. The sets of arithmetic expressions over $T$ for $T \in \mathcal{T}_N$, denoted $Expr(T)$, are defined by

$$expr ::= c | v | expr \otimes expr,$$

where $c \in T$, $v \in DV$ with $type(v) = T$, and $\otimes \in \{+,-,\ast,\div\}$. By $type(expr)$ we denote the type of all the components of the expression and therefore the type of the result\(^6\). Moreover, we define $Expr(\mathcal{T}) = \bigcup_{T \in \mathcal{T}_N} Expr(T)$.

The set of boolean expressions over $DV$, denoted $BoE(DV)$, is defined by

$$\beta ::= \text{true} | v \sim c | v \sim v' | expr \sim expr' | \beta \land \beta | \beta \lor \beta | \neg \beta,$$

where $v, v' \in DV$, $c \in type(v)$, $type(v') = type(v)$, $expr, expr' \in Expr(T)$, $type(expr) = type(expr')$. Similarly as in some programming languages, e.g. the Ada language.

The set of instructions over $DV$ and $DP$, denoted $Ins(DV, DP)$, is given by

$$\alpha ::= \epsilon | v ::= c | v ::= p | v ::= v' | v ::= expr | \alpha \alpha,$$

where $\epsilon$ denotes the empty sequence, $v, v' \in DV$, $c \in type(v)$, $p \in DP$ and $type(p) = type(v)$, $type(v') = type(v)$, $expr \in Expr(T)$, and $type(expr) = type(v)$\(^7\). Thus, an instruction over $DV$ is either an atomic instruction over $DV$ which can be either non-parametric ($v ::= c$, $v ::= v'$, $v ::= expr$) or parametric ($v ::= p$), or a (possibly empty) sequence of atomic instructions. Moreover, by $Ins^2(DV, DP)$ we denote the set consisting of all these $\alpha \in Ins(DV, DP)$ in which any $v \in DV$ appears on the left-hand side of “:=” (i.e. is assigned a new value, possibly taken from a parameter) at most once.

By a variables valuation we mean a total mapping $v : DV \rightarrow \mathcal{T}$ satisfying $v(v) \in type(v)$ for each $v \in DV$. We extend this mapping to expressions of $Expr(T)$ in the usual way. Similarly, by a parameters valuation we mean a total mapping $p : DP \rightarrow \mathcal{T}$ satisfying $p(p) \in type(p)$ for each $p \in DP$. Moreover, we assume that the domain of values for each variable and each parameter is finite. The satisfaction relation ($\models$) for a boolean expression $\beta \in BoE(DV)$ and a valuation $v$ is defined as:

$$v \models \text{true},$$

$$v \models \beta_1 \land \beta_2 \iff v \models \beta_1 \text{ and } v \models \beta_2,$$

$$v \models v \sim c \iff v(v) \sim c,$$

$$v \models expr \sim expr' \iff v(expr) \sim v(expr').$$

\(^5\)Similarly as in some programming languages, e.g. the Ada language.

\(^6\)Using different numeric types in the same expression is not allowed. The “/” operator denotes either the integer division or the “ordinary” division, depending on the context.

\(^7\)Distinguishing between assigning an arithmetic expression, and separately assigning a parameter, a constant or a variable follows from the fact that arithmetic expressions are defined for numeric types only. The same applies to the definition of boolean expressions.
Given a variables valuation $v$, a parameter valuation $p$ and an instruction $\alpha \in Ins(DV, DP)$, we denote by $v(\alpha, p)$ a valuation $v'$ such that

- if $\alpha = e$, then $v' = v$,
- if $\alpha = (v := c)$, then for all $v' \in DV$ we have $v'(v') = c$ if $v' = v$, and $v'(v') = v(v')$ otherwise,
- if $\alpha = (v := v_1)$, then for all $v' \in DV$ we have $v'(v') = v_1$ if $v' = v$, and $v'(v') = v(v')$ otherwise,
- if $\alpha = (v := expr)$, then for all $v' \in DV$ we have $v'(v') = expr$ if $v' = v$, and $v'(v') = v(v')$ otherwise,
- if $\alpha = (v := p)$, then for all $v' \in DV$ we have $v'(v') = p(p)$, and $v'(v') = v(v')$ otherwise,
- if $\alpha = \alpha_1 \alpha_2$, then $v' = (v(\alpha_1, p))(\alpha_2, p)$.

Let $X = \{x_1, \ldots, x_n, x\}$ be a finite set of real-valued variables, called clocks. The set of clock constraints over $X$, denoted $C_X(X)$, is defined by the following grammar:

$$cc ::= true \mid x_i \sim c \mid x_i - x_j \sim c \mid cc \land cc,$$

where $x_i, x_j \in X$, $c \in \mathbb{R}$, and $\sim \in \{\leq, <, =, >, \geq\}$. Let $X^+$ denote the set $X \cup \{x_0\}$, where $x_0 \notin X$ is a fictitious clock representing the constant $0$. An assignment over $X$ is a function $\alpha : X \rightarrow X^+$. $Asg(X)$ denotes the set of all the assignments over $X$.

By a clock valuation we mean a total mapping $c : X \rightarrow \mathbb{R}_+$. The satisfaction relation ($\models$) for a clock constraint $cc \in C_X(X)$ and a clock valuation $c$ is defined as

- $c \models true$,
- $c \models (x_i - x_j \sim c) \text{ iff } c(x_i) - c(x_j) \sim c$,
- $c \models (x_i \sim c) \text{ iff } c(x_i) \sim c$,
- $c \models cc_1 \land cc_2 \text{ iff } c \models cc_1$ and $c \models cc_2$.

In what follows, the set of all the clock valuations satisfying a clock constraint $cc$ is denoted by $[cc]$. Given a clock valuation $c$ and $\delta \in \mathbb{R}_+$, by $c + \delta$ we denote a clock valuation $c'$ such that $c'(x) = c(x) + \delta$ for all $x \in X$. Moreover, for a clock valuation $c$ and an assignment $\alpha \in Asg(X)$, by $c(\alpha)$ we denote a clock valuation $c'$ such that for all $x \in X$ it holds $c'(x) = c(\alpha(x))$ if $\alpha(x) \in X$, and $c'(x) = 0$ otherwise (i.e., if $\alpha(x) = x_0$). Finally, by $c^0$ we denote the initial clock valuation, i.e., the valuation such that $c^0(x) = 0$ for all $x \in X$.

**Definition 5.1.** Let $T$ be a set of discrete types. A timed automaton with discrete data and parametric assignments (TADDPA) is a tuple $A = (L, L_0, DV, DP, X, \mathcal{E}, \mathcal{I}_c, \mathcal{I}_v, v^0)$, where $L$ is a finite set of labels (actions), $L$ is a finite set of locations, $L_0 \in L$ is the initial location, $DV$ is a finite set of variables (of the types in $T$), $DP$ is a finite set of parameters (of the types in $T$), $X$ is a finite set of clocks, $\mathcal{E} \subseteq L \times L \times BoE(DV) \times C_X(X) \times Ins^0(DV, DP) \times Asg(X) \times L$ is a transition relation, $\mathcal{I}_c : L \rightarrow C_X(X)$ and $\mathcal{I}_v : L \rightarrow BoE(DV)$ are, respectively a clocks’ and a variables’ invariant functions, and $v^0 : DV \rightarrow T$ s.t. $v^0 \models \mathcal{I}_c(l^0)$ is an initial valuation of variables.
The invariant functions assign to each location a clock constraint and a boolean expression specifying the conditions under which \( A \) can stay in this location. Each element \( t = (l, I, \beta, cc, \alpha, a, l') \in \mathcal{E} \) denotes a transition from the location \( l \) to the location \( l' \), where \( l \) is the label of the transition \( t \), \( \beta \) and \( cc \) define the enabling conditions for \( l \), \( \alpha \) is the instruction to be performed, and \( a \) is the clock assignment. Moreover, for a transition \( t = (l, I, \beta, cc, \alpha, a, l') \in \mathcal{E} \) we write \( \text{source}(t) \), \( \text{label}(t) \), \( \text{vguard}(t) \), \( \text{cguard}(t) \), \( \text{instr}(t) \), \( \text{asgn}(t) \) and \( \text{target}(t) \) for \( l, I, \beta, cc, \alpha \) and \( a \) and \( l' \), respectively.

**Example 6.** Figure 1 shows a timed automaton with discrete data and parametric assignments, with \( \mathcal{L} = \{ \text{move} \} \), \( L = \{ l1, l2 \} \), \( l0 = l1 \), \( \mathcal{X} = \{ x \} \), \( DV = \{ v1, v2 \} \) and the initial values of the variables (of the integer type) given by \( v1 = 1 \), \( v2 = 2 \). The automaton can be in the location \( l1 \) under the condition \( x < 3 \), and in \( l2 \) under the condition \( v2 < 10 \). The action labelled by \text{move} \) can be taken if \( x < 5 \) and \( v1 < v2 \), and performing it results in setting the clock \( x \) to 0, increasing \( v1 \) by 10, and assigning \( v2 \) the value stored in the parameter \( p \).

The semantics of the above automata is given as follows:

**Definition 5.2.** The semantics of a TADDA \( A = (\mathcal{L}, L, l0, DV, DP, \mathcal{X}, E, I_c, I_v, v0) \) for a parameter valuation \( p : DP \to T \) is a labelled transition system\(^8\) \( S(A, p) = (Q, q0, L_S, \to) \), where \( Q = \{ (l, v, c) | l \in L \land \forall v \in DV. v(\epsilon) \in type(v) \land c \in \mathbb{R}^{\mathcal{X}} \land c \models I_c(l) \land v \models I_v(l) \} \) is the set of states, \( q0 = (l0, v0, c0) \) is the initial state, \( L_S = L \cup \mathbb{R}_+ \) is the set of labels, \( \to \subseteq Q \times L_S \times Q \) is the smallest transition relation defined by:

- for \( l \in \mathcal{L}, (l, v, c) \to (l', v', c') \) iff there exists a transition \( t = (l, l, \beta, cc, \alpha, a, l') \in \mathcal{E} \) such that \( v \models I_v(l), c \models I_c(l), v \models \beta, c \models cc, v' = v(\alpha, p) \models I_v(l'), \) and \( c' = c(a) \models I_c(l') \) (action transition),

- for \( \delta \in \mathbb{R}_+, (l, v, c) \to (l, v, c + \delta) \) iff \( c + \delta \models I_c(l) \) (time transition).

A transition \( t \) is enabled at a state \( l, v, c \) for a given parameter valuation \( p \) if \( v \models vguard(t) \), \( c \models cguard(t) \), \( c(asgn(t)) \models I_c(target(t)) \), and \( v(instr(t), p) \models I_v(target(t)) \). Intuitively, in the initial state all the variables are set to their initial values, and all the clocks are set to zero. Then, being in a state \( q = (l, v, c) \) the system can either execute an enabled transition \( t \) and move to the state \( q' = (l', v', c') \) where \( l' = target(t) \), the valuation of variables is changed according to \( instr(t) \) and the parameter valuation \( p \), and the clock valuation is changed according to \( asgn(t) \), or move to the state \( q' = (l, v, c + \delta) \) which results from passing some time \( \delta \in \mathbb{R}_+ \) such that \( c + \delta \models inv(l) \).

We say that a location \( l \) (a variables valuation \( v \), respectively) is reachable if some state \( (l, \cdot, \cdot) \) ((\cdot, v, \cdot), respectively) is reachable in \( S(A, p) \). Given \( D \subseteq DV \), a partial variables valuation \( v_D : D \to T \) is reachable if some state \( (\cdot, v, \cdot) \) s.t. \( v | D = v_D \) is reachable in \( S(A, p) \).

---

\(^8\)By a labelled transition system we mean a tuple \( S = (S, s0, \Lambda, \to) \), where \( S \) is a set of states, \( s0 \in S \) is the initial state, \( \Lambda \) is a set of labels, and \( \to \subseteq S \times \Lambda \times S \) is a (labelled) transition relation.
5.2. SAT-Based Reachability Checking

In the paper [31] we showed how to test reachability for timed automata with discrete data (TADD) using a SAT-based bounded model checking method. The main idea consist in discretising the set of the clock valuations of the automaton considered, in order to obtain a countable state space. Next, the transition relation of the transition system obtained is unfolded up to some depth $k$, and the unfolding is encoded as a propositional formula. The property to be tested is encoded as a propositional formula as well, and satisfiability of the conjunction of these two formulas is checked using a SAT-solver. Satisfiability of the conjunction allows to conclude that a path from the initial state to a state satisfying the property was found.

The automata considered in [31] are extended in this paper in the following way:

- values of the discrete variables are not only integers, but can be of several discrete types,
- arithmetic expressions used can be of a more complex form,
- the invariant function involves not only clock comparisons, but also boolean expression over values of discrete variables,
- the definition of the automaton contains additionally a set of parameters, and the set of the instructions can contain assignments of the form $a_{variable} := a_{parameter}$.

As it is easy to see, discretisation of the set of clock valuations for TADDPA can be done analogously as in [31]. The way of extending arithmetic operations on integers is described in [30]. New data types can be handled by conversions to integers; introducing extended invariants is straightforward. The only problem whose solution cannot be easily adapted from the previous approach is that SAT-based reachability testing for TADDPA involves also searching for a parameter valuation for which a state satisfying a given property can be reached. However, the idea of doing this can be derived from the idea of SAT-solvers: a SAT-solver searches for a valuation of propositional variables for which a formula holds. Thus, we represent the values of parameters by sets of propositional variables; finding a valuation for which a formula $\gamma$, which encodes that a state satisfying a given property is reachable along a path of a length $k$ implies also finding an appropriate valuation of parameters occurring along the path considered.

5.3. SAT-Based Service Composition

In order to apply the above verification method to automatic searching for sequences of concrete services able to satisfy the user’s request we translate paths of the abstract graph to timed automata with discrete data and parametric assignments. The translation uses the descriptions of concrete services, as well as the user’s query.

Consider a path $\pi = w_0 \rightarrow w_1 \rightarrow \ldots w_n$ ($n \in \mathbb{N}$) in the abstract graph, such that $w_0 \in V_p$ (i.e., it is an initial world) and $w_n \in V_k$ (i.e., it is a final world) - i.e., a sequence of worlds and abstract services which transform them. Let $O_\pi$ be the set of all the objects which occur in all the worlds along this path (i.e., $O_\pi = \{ o \in w_i \mid i = 0, \ldots, n \}$). Then, we define $V(\pi) = \{objectName.attributeName \mid objectName \in O_\pi\}$, $V_{pre}(\pi) = \{objectName.attributeName.pre \mid objectName \in O_\pi \land \exists i \in \{0, \ldots, n-1\} objectName \in w_i \cap w_{i+1}\}$, and $V'(\pi) = V(\pi) \cup \{v.isConst \mid v \in V(\pi)\} \cup \{v.isSet \mid v \in V(\pi)\} \cup \{v.isAny \mid v \in V(\pi)\} \cup V_{pre}(\pi)$. The set of the discrete variables of the automaton $A(\pi)$
corresponding to $\pi$ is equal to $V' (\pi)$. The intuition behind this construction is that for each attribute of each object occurring along the path we define a variable aimed at storing the value of the attribute $(\text{ObjectName}.\text{attributeName})$. Moreover, for such a variable we introduce three new boolean variables: the first one saying whether the flag $\text{const}$ for the attribute has been set $(\text{ObjectName}.\text{attributeName}.\text{isConst})$, the second one to express that the attribute has been set (has a nonempty value; $\text{ObjectName}.\text{attributeName}.\text{isSet}$, and the third one to specify that the value of the attribute is nonempty, but its exact value is not given $(\text{ObjectName}.\text{attributeName}.\text{isAny})$. The variables in $V_{\text{pre}}(\pi)$ (of the form $\text{ObjectName}.\text{attributeName}.\text{pre}$) are aimed at storing the values of the attributes from a pre-world of a service.

The initial values of the variables are taken from the initial world $w_0$ resulting from the user’s query:

- for each attribute $x.y$ which according to the query has a concrete value $\gamma$ in $w_0$, we set $x.y := \gamma$, $x.y.\text{isAny} := \text{false}$ and $x.y.\text{isSet} := \text{true}$; concerning $x.y.\text{isConst}$ we set it $\text{true}$ if such a condition occurs in the query, otherwise it is set to $\text{false}$,

- for each attribute $x.y$ which according to the query is set, but its value is not given directly, we set $x.y.\text{isSet} := \text{true}$, and $x.y.\text{isAny} = \text{true}$; $x.y.\text{isConst}$ is set according to the query as above; $x.y$ can obtain any value of the appropriate type (we can assume it gets a “zero” value of $\text{type}(x.y)$),

- for each attribute $x.y$ which does not occur in the query or is specified there as having the empty value we set $x.y.\text{isSet} = \text{false}$, $x.y.\text{isAny} = \text{true}$, $x.y.\text{isConst} = \text{false}$, the value of $x.y$ is set to an arbitrary value as above,

- each variable of the form $x.y.\text{pre}$ is assumed to have a “zero” value of $\text{type}(x, y)$.

**Example 7.** Consider the ontology discussed in Examples 1-3, and the query from Example 4. One of the sequences of services $\pi$ which possibly can lead to satisfying the query is $\text{SelectWare}$ and then $\text{FruitSelling}$ and then $\text{MakingJuice}$ (see Example 5). The variables of the automaton $A(\pi)$ and their types are: $f.\text{id}$ : integer, $f.\text{name}$ : $\text{FruitTypes}$, $f.\text{owner}$ : $\text{OwnerNames}$, $f.\text{capacity}$ : float, $j.\text{id}$ : integer, $j.\text{name}$ : $\text{FruitTypes}$, $j.\text{owner}$ : $\text{OwnerNames}$, $j.\text{capacity}$ : float plus the corresponding boolean variables of the form $x.y.\text{isSet}$, $x.y.\text{isAny}$ and $x.y.\text{isConst}$. Moreover, we introduce one additional variable $f.\text{owner}.\text{pre}$ : $\text{OwnerNames}$ to store the previous value of $f.\text{owner}$. All the variables of the form $x.y$ are initialised to zero values of the appropriate types, each $x.y.\text{isSet}$ and $x.y.\text{isConst}$ is initialised with $\text{false}$, and each $x.y.\text{isAny}$ is initialised with $\text{true}$.

Define for each $w_i$, $i = 0, \ldots, n$, a new location of $A(\pi)$, denoted for simplicity $w_i$ as well, and consider an edge $w_i \rightarrow w_{i+1}$ of $\pi$ ($i \in \{0, \ldots, n-1\}$), corresponding to an abstract service $s_\pi$. For each concrete service $s$ of the type of $s_\pi$ we introduce a new location $w^s_i$ and the transitions $w^s_i \xrightarrow{s} w^s_i$ and $w^s_i \xrightarrow{s} w^s_{i+1}$ (where $\varepsilon$ is an “empty” label). Then, we make use of the description of $s$ as follows (see Fig. 2 for an illustration):

- the precondition of $s$ becomes the guard of the transition $w_i \xrightarrow{s} w^s_i$ (notice that a disjunctive form is here allowed). The guard is built as follows:

\footnote{We apply the optimisation allowing to add the variables of the form $x.y.\text{pre}$ only if there is a service which refers both to $\text{pre}(x, y)$ and to $\text{post}(x, y)$, see page 195.}

\footnote{We assume here that the postcondition of $s$ contains no disjunctions; otherwise we treat $s$ as a number of concrete services each of which has the postcondition corresponding to one part of the DNF in the original postcondition of $s$.}
Figure 2. The idea of the translation, illustrated on a concrete service FruitShop with the precondition isSet(f.id) and isSet(f.owner) and f.owner=FruitShop, mustSet = (f.owner, f.capacity) and the postcondition f.capacity<100

- each predicate of the form isSet(x.y) is transformed to x.y.isSet = true,
- each predicate of the form isConst(x.y) is transformed to x.y.isConst = true,
- each predicate of the form $x.y \# z$ with $\# \in \{<,\leq,=,\geq,>\}$ and z being a constant or an expression is transformed to $x.y#z \land x.y.isSet = true \land x.y.isAny = false$; moreover, if z refers to the value of an attribute a.b, then the above condition is conjuncted with $a.b.isSet = true \land a.b.isAny = false$, for each attribute a.b occurring in z;

- the list requires of s is used to construct the instruction $\alpha$ “decorating” $w_i \xrightarrow{s} w_i^*$: initially $\alpha$ is set to $\epsilon$, then, for each attribute y of an object x occurring in requires for which it holds $x.y.isSet = true$, $\alpha$ is extended by concatenating $x.y.pre := x.y$,

- the lists mustSet and mustSetConst of s are used to construct the instruction $\alpha$ as well: for each attribute x.y occurring in the list mustSet $\alpha$ is extended by concatenating $x.y.isSet := true$, and for each attribute x.y occurring in the list mustSetConst of s $\alpha$ is extended by concatenating $x.y.isConst := true$,

- the postcondition is used as follows (x.y denotes an attribute):
  
  - the predicates Exists are ignored,
  
  - the (possibly negated) predicates of the form isSet(x.y) or isConst(x.y) result in extending the instruction $\alpha$ “decorating” $w_i \xrightarrow{} w_i^*$ by concatenating respectively $x.y.isSet := true$ or $x.y.isConst := true$ if such an instruction has not been added to $\alpha$ before (or respectively $x.y.isSet := false$ or $x.y.isConst := false$ if the predicates are negated)\\footnote{Possible inconsistencies, i.e. an occurrence of x.y in mustSet and the predicate not isSet(x.y) in postCondition, are treated as ontology errors.}. Moreover, in the
case of extending \( \alpha \) with \( x.y.isSet := false \) we extend it also by concatenating \( x.y.isAny := true \),

- each predicate of the form \( x.y = z \) or \( \text{post}(x.y) = z \) (where \( z \) can be either a concrete value or an expression\(^{12} \)) results in extending the instruction \( \alpha \) by concatenating \( x.y := z, x.y.isSet := true \) (if it has not been added before) and \( x.y.isAny := false \),

- for each predicate of the form \( x.y \# z \) or \( \text{post}(x.y) \# z \) with \( \# \in \{<, >, \leq, \geq \} \) (where \( z \) is either a concrete value or an expression) we introduce a new parameter \( p \), extend \( \alpha \) by concatenating \( x.y := p, x.y.isSet := true \) (if it has not been added before) and \( x.y.isAny := false \), and conjunct the invariant of \( w_i \)s (initially \( true \)) with the above predicate,

- if the expression \( z \) in a predicate of the form \( x.y = z \) or \( x.y \# z \) considered above refers to the value of the attribute \( a.b \) it has in the input world of \( s \), then the guard of the transition \( w_i \xrightarrow{\epsilon} w_i^s \) is conjuncted with \( a.b.isSet = true \land a.b.isAny = false \) (if such a conjunct has not been added before); if it refers to the value of the attribute \( a.b \) it has in the output world of \( s \), then the invariant of the location \( w_i^s \) is extended by conjuncting \( a.b.isSet = true \land a.b.isAny = false \) (if such a conjunct has not been added before).

- moreover, for each attribute \( x.y \) which occurs either in \( \text{mustSet} \) or in the postcondition in a predicate \( \text{isSet}(x.y) \), but does not have in the \( \text{postCondition} \) any “corresponding” predicate which allows to set its value, we introduce a new parameter \( p_{x.y}^s \), and extend \( \alpha \) by adding \( x.y := p_{x.y}^s \) and \( x.y.isAny := false \).

The invariants of \( w_i \) and \( w_{i+1} \), as well as the guard of the transition labelled with \( \epsilon \) are set to \( true \). The set of instructions of the latter transition is empty. The set of clocks of \( A(\pi) \) is empty as well.

The intuition behind the above construction is as follows: initially, only the variables of the form \( x.y \) corresponding to attributes specified by the user’s query as having concrete values are set, while the rest stores random values (which is expressed by \( x.y.isAny = true \)). Next, concrete services modify values of the variables. If the description of a service specifies that an attribute is set and specifies the exact value assigned, then the transition corresponding to execution of this service sets the corresponding variable in an appropriate way. If the exact value of the attribute set is not given, then a parameter for the value assigned is introduced, and possible conditions on this parameter (specified in the postcondition) are assigned to the target location as a part of its invariant. Moreover, before introducing any changes to the values of the variables corresponding to the attributes of the objects in the list \( \text{requires} \) their previous values are memorised.

The correctness of this translation is discussed in the appendix.

**Example 8.** Consider the automaton \( A(\pi) \) discussed in Example 7, and the concrete services listed in Example 3.

- The FruitNetMarket generates two transitions (together with the intermediate locations and the “\( \epsilon \)-transitions” outgoing them). The first one is decorated with \( f.name.isSet := true; f.owner.isSet := true; f.name := \text{strawberry}; f.owner := \text{shop1}; f.owner.isAny := false; f.name.isAny := false \), the second one is decorated in a similar way, but with \( f.name := \text{blueberry} \), the edges for the three offers of FruitNetOffers are similar.

\(^{12}\)Recall that the expressions can refer only to values the variables have in the pre-world of a service.
...the fruitselling services correspond to the following edges and intermediate locations:

- for Shop1: the guard of the first edge is $f.id.isSet = false \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet = true; f.id.isAny := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p^F_{f.own}; f.id := p^F_{f.id}; f.capacity := p^F_{f.cap}$ (where $p^F_{f.own}$ are parameters), the invariant of the intermediate location is $\neg (f.owner.pre = f.owner) \land f.id > 0 \land f.capacity > 0 \land f.capacity \leq 10$;

- for Shop2: the guard of the first edge is $f.id.isSet = false \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet := true; f.id.isAny := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p^F_{f.own}; f.id := p^F_{f.id}; f.capacity := p^F_{f.cap}$, the invariant of the intermediate location is $\neg (f.owner.pre = f.owner) \land f.id > 0 \land f.capacity > 0$,

- for Shop3: the guard of the first edge is $f.id.isSet = false \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet := true; f.id.isAny := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p^F_{f.own}; f.id := p^F_{f.id}; f.capacity := p^F_{f.cap}$, the invariant of the intermediate location is $\neg (f.owner.pre = f.owner) \land f.id > 0 \land f.capacity > 0$,

- for the services making juice from fruits:

- for HomeJuiceMaking: the guard of the first edge is $f.id.isSet = true \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet = true \land f.capacity > 0 \land f.capacity \leq 10 \land \neg (f.name = plum)$ the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p^M_{j.id}; j.capacity := f.capacity; j.name := f.name; j.owner := f.owner$ the invariant of the intermediate location is $j.capacity > 0$

- for GrandmaKitchen: the guard of the first edge is $f.id.isSet = true \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet = true \land f.capacity > 0 \land f.capacity \leq 5$, the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.capacity.isSet := true; j.capacity.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p^M_{j.id}; j.capacity := f.capacity; j.name := f.name; j.owner := f.owner$ the invariant of the intermediate location is $j.capacity > 0$

- for JuiceTex: the guard of the first edge is $f.id.isSet = true \land f.name.isSet = true \land f.owner.isSet = true \land f.capacity.isSet = true \land f.capacity > 0$ the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.capacity.isSet := true; j.capacity.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p^M_{j.id}; j.capacity := 2 \ast f.capacity; j.name := f.name; j.owner := f.owner$ the invariant of the intermediate location is $j.capacity > 0$

The above construction can be optimised in several ways. Firstly, one can add a new "intermediate" location $w_i^j$ only in the case when no location, corresponding to a service of the same type as $s$ and having the appropriate invariant, has been added before; otherwise, the transition outgoing $w_i$ can be redirected to the existing location. Secondly, the variables of the form $x.y.pre$ can be introduced only for these attributes for which there is a postcondition of a service which refers both to $pre(x).y$ and $post(x).y$. Finally, if we have several concrete services of a given type $t$ occurring as the $i$-th service along the abstract path, and - according to the above construction - need to introduce for each of them...
In order to test whether obtaining a sequence of concrete services leading to satisfying the user’s request is possible, the query is transformed to a boolean formula involving the variables of $A(\pi)$. Its satisfiability is then checked using a SAT-solver. The result contains a sequence of concrete services to be run as well as values of parameters for which executing the sequence is possible.

**Example 9.** The query from Example 4 is translated to $\neg j.\text{id}.\text{isSet} \land \neg j.\text{id}.\text{isAny} \land j.\text{capacity}.\text{isSet} \land \neg j.\text{capacity}.\text{isAny} \land j.\text{owner}.\text{isSet} \land \neg j.\text{owner}.\text{isAny} \land j.\text{id} > 0 \land j.\text{capacity} = 10 \land j.\text{owner} = \text{Me}$. In practice, we extend it by adding an additional proposition which is true in the locations $w_0, \ldots, w_3$ to avoid obtaining paths which finish before both the transitions corresponding to a concrete service are executed. After running our preliminary implementation, we have obtained the path corresponding to selling 10 units of blueberries by Shop1 and processing them by HomeJuiceMaking (see Fig. 3).

### 6. Experimental Results

The method described above has been implemented and integrated with the abstract planner as a single toolset, equipped with a user friendly graphical interface (see Fig. 3).

![Figure 3. Visualization of TADDPA for the Getting Juice example. The resulting path is marked dark grey.](image)

The tool generates the input automaton together with the reachability property to be tested, using only the ontology and the user queries. The preliminary implementation was tested on a Getting Juice example from [13, 19] as well as on Publishing Services, Medical Services, and the Travel Agency examples described in [8]. Below, we provide a brief description of these examples.
The **Publishing Services** example models the environment of services related to publishing, including several aspects of preparing publications: in the ontology there are services providing contents, photos, typesetting, publishing in the Internet, printing, etc., as well as objects: books, servers, web sites. The system (toolset) implementing our approach allows to combine different types of services registered in a repository to get the objects required. The services can have different geographical locations and run simultaneously - for instance, a web site can be created in cooperation of a development team from Poland, an American photographer and a Chinese graphic, and hosted by a provider from New Zealand. Based on a user’s request, the system chooses appropriate types of services (according to the information from the repositories) and arranges them in a correct order. Next, it determines possible ways of solving the problem, taking into account the features of the real instances of these services, stored in the repositories. We have tested two queries: the first one asking for providing a web site, specifying some its features: `select s:Site such that isSet(s.id) and isSet(s.ip_address) and s.layout="Advanced" and s.code="HTML+JavaScript"`, and the second one asking for providing a book: `select b:Book such that b.location=“Warsaw” and isSet(b.layout) and b.quality=“High” and b.quantity=100`. The results for concretising a selected branch of the abstract plan generated for the given query are presented in Tables 1 and 2. The tests have been performed using a PC equipped with Intel Core 2 Duo (2.4 GHz) and 2.3 GB RAM running Linux OS.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>120 (60)</td>
<td>2.53</td>
<td>0.20</td>
<td>61 / 108</td>
<td>10.01</td>
<td>13.3</td>
<td>1.25</td>
<td>13.99</td>
</tr>
<tr>
<td>240 (120)</td>
<td>3.07</td>
<td>0.56</td>
<td>121 / 228</td>
<td>19.21</td>
<td>14.2</td>
<td>2.06</td>
<td>24.90</td>
</tr>
<tr>
<td>600 (300)</td>
<td>4.71</td>
<td>0.79</td>
<td>301 / 588</td>
<td>46.83</td>
<td>17.1</td>
<td>3.41</td>
<td>55.74</td>
</tr>
<tr>
<td>900 (450)</td>
<td>5.69</td>
<td>2.32</td>
<td>451 / 888</td>
<td>69.95</td>
<td>19.3</td>
<td>3.99</td>
<td>81.95</td>
</tr>
<tr>
<td>1200 (600)</td>
<td>6.56</td>
<td>2.48</td>
<td>601 / 1188</td>
<td>93.34</td>
<td>22.3</td>
<td>5.17</td>
<td>107.55</td>
</tr>
</tbody>
</table>

**Table 1.** The experimental results of a SAT-based concrete composition phase for the Publishing Services example and the query requesting a web site. The abstract composition of the depth 6 was computed in fractions of seconds, regardless of the number of concrete services. The columns (from left to right) are: **Services** specifying the total number of concrete services in the ontology (and the number of services analysed in the processed plan), **Load** and **Generate** giving respectively the time of loading and parsing the ontology and the time used to generate an automaton, **States/trans** specifying the numbers of states and transitions in the resulting automaton, **BMC [s]** and **BMC [MB]** containing respectively the time and the memory consumed by the BMC module, **SAT** giving the time consumed by the SAT-solver, and **Total** providing the total composition time measured in seconds.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>120 (40)</td>
<td>2.59</td>
<td>0.03</td>
<td>41 / 72</td>
<td>4.10</td>
<td>7.9</td>
<td>0.35</td>
<td>7.01</td>
</tr>
<tr>
<td>240 (80)</td>
<td>3.17</td>
<td>0.09</td>
<td>81 / 152</td>
<td>7.58</td>
<td>8.4</td>
<td>0.58</td>
<td>11.42</td>
</tr>
<tr>
<td>600 (200)</td>
<td>4.91</td>
<td>0.56</td>
<td>201 / 392</td>
<td>18.32</td>
<td>9.7</td>
<td>1.12</td>
<td>24.91</td>
</tr>
<tr>
<td>900 (300)</td>
<td>5.78</td>
<td>0.95</td>
<td>301 / 592</td>
<td>27.25</td>
<td>11.0</td>
<td>1.76</td>
<td>35.74</td>
</tr>
<tr>
<td>1200 (400)</td>
<td>6.77</td>
<td>1.25</td>
<td>401 / 792</td>
<td>36.0</td>
<td>12.0</td>
<td>2.18</td>
<td>46.20</td>
</tr>
</tbody>
</table>

**Table 2.** The results for the Publishing Services example: `select b:Book such that b.location=“Warsaw” and isSet(b.layout) and b.quality=“High” and b.quantity=100`. The meaning of the column captions is the same as in Table 1; the length of the abstract plan concretised was 4.
The Travel Agency example illustrates the use of abstract and concrete planning to create a service that models a travel agency offering excursions with additional events. Service providers can register the following types of services: transport companies expecting a destination and date constraints, and returning offers containing available places, departure and arrival dates as well as a price, companies offering accommodation, expecting the user to specify a location, type of a room (single, double, etc.) and answering with offers specifying locations and prices, and agencies offering excursions to local points of interest, to be queried about availability and prices. The user specifies a destination and an origin, a start date and a duration of his trip, and the number of persons involved in traveling. Additionally, he can limit the number of transport segments (flights, bus trips. The user may also state that he wants to use local attractions at destinations visited during the trip. He can also specify the numbers and types of the services requested - e.g., he may want to see four National Parks while staying at a particular place. The query tested was select t:Trip such that t.no_of_people=2 and t.end_location = “Budapest” and t.start_location=“Warsaw” and isSet(t.returned). The results for a selected abstract scenario are shown in Table 3.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>70 (100)</td>
<td>2.27</td>
<td>0.29</td>
<td>107 / 192</td>
<td>19.86</td>
<td>16.7</td>
<td>2.61</td>
<td>25.03</td>
</tr>
<tr>
<td>140 (200)</td>
<td>2.73</td>
<td>0.64</td>
<td>191 / 360</td>
<td>35.96</td>
<td>18.8</td>
<td>3.80</td>
<td>43.13</td>
</tr>
<tr>
<td>350 (500)</td>
<td>3.44</td>
<td>1.73</td>
<td>453 / 884</td>
<td>84.22</td>
<td>25.3</td>
<td>8.12</td>
<td>97.51</td>
</tr>
<tr>
<td>525 (750)</td>
<td>4.38</td>
<td>3.69</td>
<td>678 / 1334</td>
<td>131.2</td>
<td>32.3</td>
<td>14.52</td>
<td>153.79</td>
</tr>
<tr>
<td>700 (1000)</td>
<td>4.74</td>
<td>7.09</td>
<td>903 / 1784</td>
<td>175.1</td>
<td>38.1</td>
<td>19.78</td>
<td>206.71</td>
</tr>
</tbody>
</table>

Table 3. The results for the Travel Agency example: select t:Trip such that t.no_of_people=2 and t.end_location = “Budapest” and t.start_location=“Warsaw” and isSet(t.returned); concretising a plan of the length 10. The meaning of the column captions is the same as in Table 1.

The Medical services example illustrates an application of the framework in the area of medical services. A user (being a patient) can plan healthcare services offered by clinics dealing with a wide range of medical services, as well as by more specialized providers. He can be examined by a general practitioner or a specialist, undergo simple or complex laboratory tests, and receive several medical treatments. A list of the available medical specialists, the types of the tests, and the list of treatments with appropriate characteristics are defined by the service provider in the ontology. A query expresses an intention of the patient, which is to be satisfied by a sequence of services. The results for a query tested are presented in Table 4.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>41 (21)</td>
<td>2.23</td>
<td>0.56</td>
<td>27 / 42</td>
<td>5.19</td>
<td>12.1</td>
<td>0.54</td>
<td>8.52</td>
</tr>
<tr>
<td>401 (201)</td>
<td>3.67</td>
<td>1.11</td>
<td>207 / 402</td>
<td>35.22</td>
<td>14.3</td>
<td>1.57</td>
<td>41.56</td>
</tr>
<tr>
<td>801 (401)</td>
<td>5.08</td>
<td>1.71</td>
<td>407 / 802</td>
<td>68.95</td>
<td>17</td>
<td>3.18</td>
<td>78.91</td>
</tr>
</tbody>
</table>

Table 4. The results for the medical services composition: select derma: DermatologistVisit, test: Test, therapy: Therapy such that derma.doctor_name = “Nowak” and derma.location = “Warsaw” and test.location = “Warsaw” and therapy.location = “Warsaw”; concretising the scenario of the length 5. The meaning of the column captions is the same as in Table 1.
The practical implications of the last example are that our system is currently implemented for a really-existing business environment (the Center of Rehabilitation and Cosmetology in Lodz, Poland; see [8] for details).

7. Final Remarks

This work is a part of a bigger project aimed at developing a complete system for automated composition of web services. Our approach applies a three-phase composition, the first stage of which generates an abstract plan showing sequences of service types which can potentially enable to reach the user’s goal specified in a query given. The aim of the second phase of the composition is to find concrete services of appropriate types whose individual limitations and capabilities enable to combine them in a sequence (corresponding to a branch of the abstract plan) whose execution leads to satisfying the user’s request. In this paper we have shown an approach to this stage of the composition, based on SAT-based BMC technique, i.e., testing reachability for (timed) automata with discrete data and parametric assignments. It should be noted that this stage of the composition requires collecting offers from cooperating services. This implies a kind of a conversation with repositories and web services, which is, however, beyond the scope of this work. One of possible realizations of the phase collecting offers is given in [7].

As for now, we do not make use of the timed part of the model. So, our further work involves extending the approach by testing timing dependencies, and searching for scenarios of a cost required.

References


A. Correctness of the Translation

Consider a path of the abstract graph $\pi = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_n \ (n \in \mathbb{N})$, where $w_0$ is an initial world, and $w_n$ a final world. The worlds along the path are sets of objects being sets of attributes. Each attribute can be either set or not set; moreover it can be treated as “unchangeable” (i.e., it has the flag $\text{const}$ set to $\text{true}$).

The structure of the automaton reflects the above as follows: for each attribute $y$ of an object $x$ occurring in a world along the path there are three variables aimed respectively at storing the value of the attribute $(x.y)$, the value of its $\text{const}$ flag $(x.y.\text{isConst})$ and the fact that the attribute has a nonempty value, i.e., is set $(x.y.\text{isSet})$. Due to the fact that each variable $x.y$ stores a value even if the corresponding attribute is in fact assigned no concrete one (similarly as it is in the case if uninitialised variables in programming languages), the automaton contains also (for each attribute $x.y$) the boolean variable $x.y.\text{isAny}$ whose aim is to prevent using the value of $x.y$ in such a case (the value of $x.y.\text{isAny}$ is then $\text{false}$). Moreover, the automaton contains also the variable $x.y.\text{pre}$ for each attribute $y$ of each object $x$ which occurs in both the input and the output world of a service along the path (i.e., can be an element of the list $\text{requires}$ of this service). The variable enables to refer in the postcondition of the service to the value $x.y$ had in its input world (i.e., to handle the predicates involving $\text{pre}(x).y$ and $\text{post}(x).y$).

The initial values of the variables which refer to the objects existing in $w_0$ reflect the settings in the initial world taken from the query:

- if in the initial world an attribute $x.y$ is assigned a concrete value $a$, then the initial setting of the corresponding variables is: $x.y = a$, $x.y.\text{isSet} = \text{true}$, $x.y.\text{isAny} = \text{false}$, the variable $x.y.\text{isConst}$ is also set in an appropriate way (corresponding to the contents of the query),

- if in the initial world an attribute $x.y$ is set but the query does not specify its exact value, then the initial setting of the corresponding variables is: $x.y.\text{isSet} = \text{true}$, $x.y.\text{isAny} = \text{true}$, $x.y$ gets an arbitrary value (which is not used due to the fact that in the translation pre- and postconditions are
handled in such a way that using the value of \( x.y \) involves checking whether \( x.y.is\text{Any} = false \), the variable \( x.y.is\text{Const} \) is also set in an appropriate way.

- if in the initial world the attribute \( x.y \) has the empty value (is not set), then the initial setting of the corresponding variables is \( x.y.is\text{Set} = false, x.y.is\text{Const} = false, x.y.is\text{Any} = true \) and \( x.y \) is assigned an arbitrary value (not used).

All the other variables (either corresponding to the objects which do not exist in the initial world, or to the attributes of the objects from the initial world which are not mentioned in the query) are initialised with the values expressing the fact that the attributes they correspond to are not set (\( x.y.is\text{Set} = false, x.y.is\text{Const} = false, x.y.is\text{Any} = true \), \( x.y \) has an arbitrary value).

Next, we show that the automaton reflects the behaviours of the services. We start with discussing a single service, and then discuss the topics of sequential services and of services of the same type.

Consider an abstract service \( sa_i \), an edge of an abstract scenario \( w_i \rightarrow w_{i+1} \) corresponding to \( sa_i \), and a concrete service \( sc \) of the type of \( sa_i \). Executing the service \( sc \) corresponds to moving the automaton \( A(\pi) \) from the location \( w_i \) to \( w_{i+1} \) through the intermediate location \( w_{i}^{sc} \).

- the service \( sc \) can be executed if its precondition is satisfied. The precondition of \( sc \) is transformed to the guard of the edge \( w_i \rightarrow w_{i}^{sc} \), and therefore traversing this edge is possible if this precondition holds. The rules of building the guard ensure that the value of a variable \( x.y \) is used only if it is assigned (i.e., the guard cannot be evaluated to true by reading the values of uninitialised variables),

- the objects in the list \( \text{requires} \) of \( sc \) exist both in its input world and in the output world, and can be modified by executing \( sc \). As the values of their attributes from the input world can occur in the postcondition of \( sc \), the instruction “decorating” \( w_i \rightarrow w_{i}^{sc} \) begins with the assignments \( x.y.pre = x.y \) for all the attributes of all the objects from \( \text{requires} \) which are set in the input world (i.e., have nonempty values). Placing these instructions at the beginning of the instruction ensures that the variables \( x.y.pre \) are assigned the “input” values of the attributes (and not the values assigned by the service when it executes),

- if the service \( sc \) sets an attribute \( x.y \) (which means that it occurs in \( \text{mustSet} \) or the postcondition of \( sc \) contains either the predicate \( \text{isSet}(x.y) \) or a predicate which implies that \( x.y \) is set, i.e., a predicate which compares the value of \( x.y \) with an expression), then this fact is reflected in the automaton by assigning the variable \( x.y.is\text{Set} \) the value true when the automaton moves from \( w_i \) to \( w_{i}^{sc} \),

- if the service \( sc \) makes an attribute \( x.y \) constant (which means that it occurs in \( \text{mustSetConst} \) or the postcondition of \( sc \) contains the predicate \( \text{isConst}(x.y) \)), then this fact is reflected in the automaton by assigning the variable \( x.y.is\text{SetConst} \) the value true when the automaton moves from \( w_i \) to \( w_{i}^{sc} \),

- if the service “unsets” an attribute \( x.y \) or makes it not constant (i.e., its postcondition contains the predicate \( \text{not isSet}(x.y) \) or \( \text{not isConst}(x.y) \)), then this fact is reflected in the automaton by assigning the variable \( x.y.is\text{Set} \) or respectively \( x.y.is\text{SetConst} \) the value true when the automaton moves from \( w_i \) to \( w_{i}^{sc} \). Moreover, in the case of “unsetting” \( x.y \) the variable \( x.y.is\text{Any} \) is assigned the value true as well (which says that the value stored in \( x.y \) cannot be treated as a “valid” one),
• if the service \( sc \) assigns a value to an attribute \( x.y \) then we can have the following cases:

  – if the value assigned is precisely specified in the postcondition of \( sc \) (i.e., the postcondition contains a predicate of the form \( x.y=z \) or \( \text{post}(x).y=z \), where \( z \) is a concrete value or an expression), then this fact is expressed in the automaton by assigning to \( x.y \) the expression \( z \) (obtained from \( z \) by replacing the names of attributes by the corresponding variables) when the automaton moves from \( w_i \) to \( w_i^{sc} \). Additionally, the variable \( x.y.isAny \) is assigned the value \( false \) in order to express that \( x.y \) has a “valid” value,

  – if the value assigned by the service is not specified exactly (i.e., the postcondition of \( sc \) contains no predicate of the form \( x.y=z \), but either \( x.y \) occurs in \( \text{mustSet} \), or the postcondition of \( sc \) contains the predicate \( \text{isSet}(x.y) \)), then this fact is handled in the automaton by assigning the variable \( x.y \) a parameter and the variable \( x.y.isAny \) the value \( false \) when the automaton moves from \( w_i \) to \( w_i^{sc} \). Moreover, if the postcondition of \( sc \) contains any predicate restricting the value which is assigned to \( x.y \) by \( sc \) (i.e., a predicate of the form \( x.y \# z \) with \( \# \in \{<,\leq,>,\geq\} \) and \( z \) being a constant or an expression), then this fact is reflected in the automaton by adding the corresponding condition to the invariant of the location \( w_i^{sc} \) (which prevents the automaton from entering \( w_i^{sc} \) if the condition on \( x.y \) does not hold).

Moreover, if the expression \( z \) refers to an attribute \( a.b \) then the translation should ensure that the variable \( a.b \) is read only if it is assigned a proper value. Thus, if \( z \) refers to the value of \( a.b \) in the input world of \( sc \), then the guard of the edge \( w_i \to w_i^{sc} \) is conjuncted with the condition expressing that \( a.b \) was set to a “valid” value before execution of the service (\( a.b.isSet = true \land a.b.isAny = false \)). If \( z \) refers to the value of \( a.b \) in the output world of \( sc \), then the invariant of the location \( w_i^{sc} \) is conjuncted with the condition expressing that \( a.b \) has a “valid” value in the output world of the service, i.e., \( a.b.isSet = true \land a.b.isAny = false \) (notice that we do not need to require the service to set \( a.b \), so the instruction decorating the edge \( w_i \to w_i^{sc} \) is not modified).

Moving from \( w_i^{sc} \) to \( w_{i+1} \) is done unconditionally.

The predicates \( \text{Exists} \) are ignored due to the fact that the (non)existence of certain objects in certain worlds is handled on the level of abstract planning.

Next, consider two services \( sc_1, sc_2 \) of the type of \( sa_i \). It is easy to see from the construction that the automaton moves from \( w_i \) to \( w_{i+1} \) either through \( w_i^{sc_1} \) or through \( w_i^{sc_2} \), and due to the fact that the precondition of \( sc_k \) (\( k = 1, 2 \)) becomes the guard of \( w_i \to w_i^{sc_k} \), its postcondition builds the invariant of \( w_i^{sc_k} \), and the modifications of values of attributes made by \( sc_k \) are used to build the instruction decorating \( w_i \to w_i^{sc_k} \), then the behaviour of the automaton reflects the fact that only one service of the type of \( sa_i \) is executed as the \( i \)-th in the concrete plan. Considering sequential executions of concrete services, having the invariants of \( w_i \) (\( i = 0, \ldots, n \)) set to \( true \) corresponds to the fact that the results of execution of a concrete service create “unconditionally” a world which is “inherited” by a next service as its input one.