

Towards Automatic Composition of Web Services: A SAT-Based Phase [★]

Wojciech Penczek^{1,2}, Agata Półrola³, and Andrzej Zbrzezny⁴

¹ Polish Academy of Sciences, ICS, Ordonia 21, 01-237 Warsaw, Poland

² University of Podlasie, ICS, Sienkiewicza 51, 08-110 Siedlce, Poland
penczek@ipipan.waw.pl

³ University of Łódź, FMCS, Banacha 22, 90-238 Łódź, Poland
polrola@wmi.uni.lodz.pl

⁴ Jan Długosz University, IMCS, Armii Krajowej 13/15, 42-200 Częstochowa, Poland
a.zbrzezny@ajd.czyst.pl

Abstract. Automating the composition of web services is an object of a growing interest nowadays. In our former paper [3] 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 resulting in 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.

1 Introduction

In recent years there has been a growing interest in automating the composition of web services. The number of more and more complex Internet services is still growing nowadays; several standards describe how services can be invoked (WSDL [17]), how they exchange information (SOAP [13]), how they synchronise the executions in complex flows (BPEL [16]), and finally how they can be discovered (UDDI [15]). However, 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 fact cannot be detected automatically without unification of their interfaces made by the providers.

In our work [3] we proposed an approach to automatic composition of services which can potentially solve the above problems. The problem of automatic composition of web services is converted to the problem of building a graph of worlds consisting of

[★] Supported by the Polish National Science Center under the decision DEC-2011/01/B/ST6/01477

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, 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 the languages 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. This can also involve choosing a plan which is optimal from the user's point of view. 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 [3] described a method of generating an abstract graph of services. In the current work we deal with the second phase of composition: concretising abstract flows, i.e., with searching for sequences of concrete services which can lead to satisfying user's request. We apply model checking techniques to this aim. The substage 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. Next, in Sec. 5 we present our approach to SAT-based concretising abstract scenarios. Sec. 6 show experimental results and concluding remarks.

2 Related Work

There are many papers dealing with the topic of web services composition [4, 7–10, 14]. Some of these works consider static approaches, where flows are given as a part of the input, while the others deal with dynamically created flows. One of the most active research areas is a group of methods referred to as AI Planning [4]. Several approaches use Planning Domain Definition Language (PDDL [5]). 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 [6] 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. Another methodology is the logic-based program synthesis [8]. Definitions of web services and user requirements, specified in DAML-S [2], are translated to formulas of

Linear Logic (LL): the descriptions of web services are encoded as LL axioms, while a requirement is specified as a sequent to be proven by the axioms. Then, a theorem prover determines if such a proof exists.

Besides the automatic approaches mentioned above, there exist also half-automatic methods assuming human assistance at certain stages [12]. Some approaches are based on specifying a general plan of composition manually; the plan is then refined and updated in an automatic way.

Inspired by the Entish project [1], our approach enables to model automated composition 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.

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 then understood as a function which transforms a set of data into another set of data (or as a transition between them). The sets of data are called *worlds*. The worlds can be described by the use of an *ontology*, i.e., a formal representation of a knowledge about them.

Definition 1 (World and objects). *The universum is the set of all the objects. The objects have the following features:*

- each object is either a concrete object or an abstract object,
- each object contains named attributes whose values are either other objects or:
 - values of simple types (numbers, strings, boolean values; called simple attributes) or NULL (empty value) for concrete objects,
 - values from the set {NULL, SET, ANY} for abstract objects.
 If an attribute *A* of the object *O* is an object itself, then *O* is extended by all the attributes of *A* (of the names obtained by adding *A*'s name as a prefix). Moreover, when an object having an object attribute is created, its subobject is created as well, with all the attributes set to NULL.
- each simple attribute has a boolean-valued flag `const`.

A world is a set of objects chosen from the universum. Each object in a world is identified by a unique name.

By default each `const` flag is set to `false`. If the flag of an attribute is `true`, then performing on the object any operation (service) which sets this attribute (including services initialising it) is not allowed (the value of the attribute is considered to be final). The attributes are referred to by `ObjectName.AttributeName`.

Definition 2 (Object state, world state). *A state of an object *O* is a function V_o assigning values to all the attributes of *O* (i.e., is the set of pairs (AttributeName, AttributeValue), where AttributeName ranges over all the attributes of *O*). A state of a world is defined as the state of all the objects of this world.*

In order to reason about worlds and their states we define the following two-argument functions (the second default argument of these functions is the world we are reasoning about):

- `Exists` - a function whose first argument is an object, and which says whether the object exists in the world,
- `isSet` - a function whose first argument is an attribute of an object, and which says whether the attribute is set (has a nonempty value),
- `isConst` - a function whose first argument can be either an attribute or an object. When called for an attribute, the function returns the value of its `const` flag; when called for an object it returns the conjunction of the `const` flags of all the attributes of this object.

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* or a *metaservice*), and on the level of objects (*concrete services*). The description of a service includes, besides specifying input and output data types, also declaration of introducing certain changes to a world, i.e., of creating, removing and modifying objects. The definition of a service is as follows:

Definition 3 (Service). A service is an object of a non-abstract subclass¹ of the abstract class `Service`. A service contains (initialised) attributes, inherited from the base class `Service`. The attributes 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`).

Moreover, a service can contain a set of quality attributes.

A service modifies (transforms) a world, as well as the world's state. The world to be transformed by a service is called its *pre-world* (*input world*), while the result of the execution is called a *post-world* (*output world*). Modifying a world consists in modifying a subset of its objects. The objects being transformed by one service cannot be modified by another one at the same time (i.e., transforming objects is an atomic activity). A world consisting of a number of objects can be transformed into a new state in two ways²: by a service which operates on a subset of its elements, or by many services which operate concurrently on disjoint subsets of its elements.

The groups of attributes are presented in the definitions below.

¹ 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.

² Services which create new objects are not taken into account.

Definition 4 (Processing lists). *The processing lists are as follows:*

- `produces` - a list of named objects of classes whose instances are created by the service in the post-world,
- `consumes` - a list of named objects of classes whose objects are taken from the input world, and do not exist in the world resulting from the service execution (the service removes them from the world),
- `requires` - a list of named objects of classes whose instances are required to exist in the current world to invoke the service and are still present in the output world.

The formal parameters from the above lists define an alphabet for modification lists and validation formulas.

Definition 5 (Modification lists). *The modification lists are as follows:*

- `mustSet` - a list of attributes of objects occurring in the lists `produces` and `requires` of a service, which are obligatorily set (assigned a nonempty value) by this service,
- `maySet` - a list of attributes of objects occurring in the lists `produces` and `requires` of a service, which may (but not must) be set by this service,
- `mustSetConst` - a list of attributes of the objects which occur in the lists `produces` and `requires` of a service, which are obligatorily set as being constant in the worlds after executing this service,
- `maySetConst` - a list as above, but of the attributes which may be set as constant.

A grammar for the above lists can be found in [3]. The attributes of the objects appearing in processing lists which do not belong to the union of lists `mustSet` and `maySet` are not changed when the service is called.

Definition 6 (Validation formulas). *The validation formulas are as follows:*

- `preCondition` - a formula which describes the condition under which the service can be invoked. It consists of atomic predicates over the names of objects from the lists `consumes` and `requires` of the service and over their attributes, and is written in the language of the first order calculus without quantification (atomic predicates with conjunction, disjunction and negation connectives). The language of atomic predicates contains comparisons of expressions over attributes with constants, and functions calls with object names and attributes as arguments. In particular, it contains calls of the functions `isSet`, `isConst` and `Exists`³.
- `postCondition` - a formula which specifies conditions satisfied by the world resulting from invoking the service. The formula consists of atomic predicates over the names of objects from the lists `consumes`, `produces` and `requires` of the service and over their attributes. To the objects and attributes one can apply pseudofunctions `pre` and `post` which refer to the state of an object or an attribute in the

³ Using `Exists` in `preCondition` is redundant w.r.t. using an appropriate object in the list `consumes` or `requires`. However, the future directions of developing the service description language mentioned in the final part of the paper, include moving modification lists to validation formulas.

input and the output world of this service, respectively. By default, the attributes of objects listed in `consumes` refer to the state of the pre-world, whereas these in `produces` and `requires` - to the state of the post-world.

Definition 7. A service U is enabled (executable) in the current state of a world S if:

- each object O from the lists `consumes` and `requires` of U can be mapped onto an object in S , of the class of O or of its subclass; the mapping is such that each object in S corresponds to at most one object from the above lists;
- for the objects in S which, according to the above mapping, are actual values of the parameters in `consumes` and `requires` the formula `preCondition` of U holds,
- the list `mustSet` of U contains no attributes for which, in objects which are actual values of the parameters, the flag `const` is set.

Definition 8. A service U executable at the current world S produces a new world S' in which:

- there are all the objects from S , besides these which in the mapping done for executing U were actual values for the parameters in `consumes`,
- there is a one-to-one mapping between all the other objects in S' and the objects in the list `produces` of U , such that each object O from the list `produces` corresponds to an object in S' which is of a (sub)class of O ;
- for the objects which, according to the above mappings, are actual values of the parameters in the processing lists the formula `postCondition` holds,
- in the objects which are actual values of the appropriate parameters the flags `const` of the attributes listed in `mustSetConst` of U are set, and the attributes listed in `mustSet` of U have nonempty values,
- assuming the actual values of the parameters as above, all the attributes of all the objects existing both in S and in S' which do not occur neither in `mustSet` nor in `maySet` have the same values as in the world S ; the same holds for the flags `const` of the attributes which do not occur neither in `mustSetConst` nor in `maySetConst`. Moreover, all the attributes listed in `mustSet` or `maySet` which are of nonempty values in S , in S' are of nonempty values as well.

3.1 Concrete Services

We assume here 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 in this case).

A grammar for validation formulas is as follows:

```

<objectName> ::= <objectName from consumes> |
                pre(<objectName from requires>) |
                post(<objectName from requires>) |
                <objectName from produces>
<objectAttribute> ::= <objectName>.<attributeName> |
                    <objectName>.<objectAttribute>
<expressionElement> ::= "integer value" | "real value" |

```

```

                                <objectAttribute> "of a numeric type"
<arithmOp> ::= + | - | * | /
<expression> ::= <expressionElement> |
                                <expression> <arithmOp> <expression>
<compOp> ::= = | < | <= | > | >=
<atomicPredicate> ::= Exists(<objectName>) |
                                isSet(<objectAttribute>) |
                                isConst(<objectAttribute>) |
                                not <atomicPredicate> |
                                <objectAttribute> <compOp> <expression> |
                                <objectAttribute> <compOp> "value"
<conjunction> ::= <atomicPredicate> |
                                <atomicPredicate> and <conjunction>
<validationFormula> ::= <conjunction> |
                                <conjunction> or <validationFormula>

```

It should be noticed that `pre()` and `post()` are allowed in `postCondition` only. Moreover, in this paper we assume that the expressions involve only attributes which refer either to names of objects from `consumes`, or to the names of objects which are of the form `pre(objectName from requires)` (i.e., that the expressions involve only values of attributes in 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 type⁴.

The values of the attributes and the values occurring in comparisons in the atomic predicates above are as follows:

- boolean values,
- integer values of a certain range⁵,
- characters,
- real values of a certain range, with the precision limited to a number of decimal places (usually two),
- values of certain enumeration types.

Enumeration types are used instead of strings. In fact, 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 (e.g. names of countries, cities, names of 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 values⁶.

⁴ The grammar for 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.

⁵ A natural restriction when using programming languages.

⁶ Similarly as in the Ada programming language.

4 Abstract Planning

The aim of the composition process is to find a sequence of services whose execution can satisfy a user's goal. The user describes its goal in a declarative language defined by the ontology. He specifies (possibly partially) an initial and a final (desired) world, possibly giving also some evaluation criteria. The query is defined in the following way:

Definition 9 (Query). A query consists of the following elements:

- an initial domain - a list of named objects which are elements of the initial world. The form of the list is analogous to the form of the list `produces` in the description of a service;
- an initial clause specifying a condition which is to be satisfied by the initial world. The clause is a formula over the names of objects and their attributes, taken from the initial domain. The grammar of the clause is analogous to the grammar of the `preCondition`;
- an effect domain - a list of named objects which have to be present in a final world (i.e., a subset the final world must contain);
- an effect clause specifying a condition which is to be satisfied by the final world. The clause is a formula over the names of objects and their attributes from both the domains defined above; references to the initial state of an object, if ambiguous, are specified using the notations `pre(objectName)` and `post(objectName)`, analogously as in the language used in the formulas `postCondition` of services. The grammar of the effect clause is analogous to the grammar of the `postCondition`;
- an execution condition - a formula built over services (unknown to the user when specifying the query) from a potential run performing the required transformation of the initial world into a target world. While construction of this formula simple methods of quantification and aggregation are used;
- a quality function - a real-valued function over the initial world, the final world and services in a run, which specifies a user's criterion of valuating the quality of runs. The run of the smallest value of this function is considered to be the best one.

The last two parts of a query are used after finishing both the abstract planning phase and the first part of concrete planning, which adjusts types and analyses pre- and post-conditions of concrete services.

The aim of a 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 [3]) 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*.

The abstract graph is a directed multigraph. The nodes of the graph are worlds in certain states, while its edges are labelled by services. Notice that such a labelling carries an information which part of a input world (node) is transformed by a given service (that is specified by actual values of the parameters in `consumes` and `requires` of the service), and which part of the output world (node) it affects (the lists `produces`

and `requires` of this service). We distinguish some nodes of the graph - these which have no input edges represent alternative initial worlds, while these with no output edges are alternative final worlds. A formal definition of the abstract graph is as follows:

Definition 10. An abstract graph is a tuple $GA = (V, V_p, V_k, E, L)$, where V is a subset of the set S of all the worlds, $V_p \subseteq V$ is a set of initial nodes, $V_k \subseteq V$ is a set of final nodes, and $E \subseteq V \times V$ is a transition relation s.t. $e = (v, v') \in E$ iff $L(e)$ transforms the world v into v' , where $L : E \rightarrow U$ is a function labelling the edges with services.

5 Main Idea

From the phase of abstract composition [3] we get a graph showing the sequences of service types which can potentially lead to satisfying user's request. The next step towards obtaining a flow to be run is to find concrete services of the appropriate types whose offers enable satisfying the query. We use SAT-based bounded model checking to this aim. In the paper [19] we have shown how to test reachability for timed automata with discrete data using BMC and the model checker Verics. We adapt the above approach.

The main idea of our solution consists in translating each path of the abstract graph to a timed automaton with discrete data and parametric assignments (TADDPA). The automaton represents concrete services of appropriate types (corresponding to the types of services in the scenario we are working on) which can potentially be executed to reach the 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 assigned by a service is unknown. Next, we test reachability of a state satisfying the user's query. If such a state is reachable, we get a reachability witness, containing both an information about a sequence of concrete services to be executed to reach the goal and the values of parameters for which this sequence is executable.

In spite of using timed automata we currently do not make use of the timing part of this formalism, but the reason for using them is twofold. Firstly, doing this allowed us to adapt the existing implementation for timed automata with discrete data (modified to handle their extension - TADDPA). Secondly, in the future we are going to use the clocks to represent the declared times of services executions, which should enable us searching for scenarios of an appropriate timed length.

Below, we introduce all the elements of our approach.

5.1 Timed Automata with Discrete Data and Parametric Assignments

Given a set of discrete types $\mathcal{T} = \bigcup_{i=1, \dots, n} T_i$ ($n \in \mathbb{N}$), 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 ordering⁷ 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 $T \in \mathcal{T}$. Let DV be a finite set of variables whose types belong to \mathcal{T} , and let DP be a finite set of parameters whose

⁷ Similarly as in some programming languages, e.g. the Ada language.

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

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

where $c \in T$, $v \in DV$ with $type(v) = T$, and $\otimes \in \{+, -, *, /\}$. By $type(expr)$ we denote the type of all the components of the expression and therefore the type of the result⁸. 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

$$\beta ::= true \mid v \sim c \mid v \sim v' \mid expr \sim expr' \mid \beta \wedge \beta \mid \beta \vee \beta \mid \neg\beta,$$

where $v, v' \in DV$, $c \in type(v)$, $type(v') = type(v)$, $expr, expr' \in Expr(\mathcal{T})$, $type(expr) = type(expr')$, and $\sim \in \{=, \neq, <, \leq, \geq, >\}$.

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

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

where ϵ denotes the empty sequence, $v, v' \in DV$, $c \in type(v)$, $p \in DP$ and $type(p) = type(v)$, $type(v') = type(v)$, $expr \in Expr(\mathcal{T})$, and $type(expr) = type(v)$ ⁹. 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 $Ins^\diamond(DV, DP)$ 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 $\mathbf{v} : DV \rightarrow \mathcal{T}$ satisfying $\mathbf{v}(v) \in type(v)$ for each $v \in DV$. We extend this mapping to expressions of $Expr(\mathcal{T})$ in the usual way. Similarly, by a *parameters valuation* we mean a total mapping $\mathbf{p} : DP \rightarrow \mathcal{T}$ satisfying $\mathbf{p}(p) \in type(p)$ 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 \mathbf{v} is defined as: $\mathbf{v} \models true$, $\mathbf{v} \models \beta_1 \wedge \beta_2$ iff $\mathbf{v} \models \beta_1$ and $\mathbf{v} \models \beta_2$, $\mathbf{v} \models \beta_1 \vee \beta_2$ iff $\mathbf{v} \models \beta_1$ or $\mathbf{v} \models \beta_2$, $\mathbf{v} \models \neg\beta$ iff $\mathbf{v} \not\models \beta$, $\mathbf{v} \models v \sim c$ iff $\mathbf{v}(v) \sim c$, $\mathbf{v} \models v \sim v'$ iff $\mathbf{v}(v) \sim \mathbf{v}(v')$, and $\mathbf{v} \models expr \sim expr'$ iff $\mathbf{v}(expr) \sim \mathbf{v}(expr')$. Given a variables valuation \mathbf{v} , a parameter valuation \mathbf{p} and an instruction $\alpha \in Ins(DV, DP)$, we denote by $\mathbf{v}(\alpha, \mathbf{p})$ a valuation \mathbf{v}' such that

- if $\alpha = \epsilon$ then $\mathbf{v}' = \mathbf{v}$,
- if $\alpha = (v := c)$ then for all $v' \in DV$ it holds $\mathbf{v}'(v') = c$ if $v' = v$, and $\mathbf{v}'(v') = \mathbf{v}(v')$ otherwise,
- if $\alpha = (v := v_1)$ then for all $v' \in DV$ it holds $\mathbf{v}'(v') = v_1$ if $v' = v$, and $\mathbf{v}'(v') = \mathbf{v}(v')$ otherwise,

⁸ 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.

⁹ 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.

- if $\alpha = (v := \text{expr})$ then for all $v' \in DV$ it holds $\mathbf{v}'(v') = \text{expr}$ if $v' = v$, and $\mathbf{v}'(v') = \mathbf{v}(v')$ otherwise,
- if $\alpha = (v := p)$ then for all $v' \in DV$ it holds $\mathbf{v}'(v') = \mathbf{p}(p)$, and $\mathbf{v}'(v') = \mathbf{v}(v')$ otherwise,
- if $\alpha = \alpha_1\alpha_2$ then $\mathbf{v}' = (\mathbf{v}(\alpha_1, \mathbf{p}))(\alpha_2, \mathbf{p})$.

Let $\mathcal{X} = \{x_1, \dots, x_{n_{\mathcal{X}}}\}$ be a finite set of real-valued variables, called *clocks*. The set of *clock constraints* over \mathcal{X} , denoted $\mathcal{C}_{\mathcal{X}}(\mathcal{X})$, is defined by the grammar:

$$\mathbf{cc} ::= \text{true} \mid x_i \sim c \mid x_i - x_j \sim c \mid \mathbf{cc} \wedge \mathbf{cc},$$

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

By a *clock valuation* we mean a total mapping $\mathbf{c} : \mathcal{X} \rightarrow \mathbb{R}_+$. The satisfaction relation (\models) for a clock constraint $\mathbf{cc} \in \mathcal{C}_{\mathcal{X}}(\mathcal{X})$ and a clock valuation \mathbf{c} is defined as $\mathbf{c} \models \text{true}$, $\mathbf{c} \models (x_i \sim c)$ iff $\mathbf{c}(x_i) \sim c$, $\mathbf{c} \models (x_i - x_j \sim c)$ iff $\mathbf{c}(x_i) - \mathbf{c}(x_j) \sim c$, and $\mathbf{c} \models \mathbf{cc}_1 \wedge \mathbf{cc}_2$ iff $\mathbf{c} \models \mathbf{cc}_1$ and $\mathbf{c} \models \mathbf{cc}_2$. In what follows, the set of all the clock valuations satisfying a clock constraint \mathbf{cc} is denoted by $\llbracket \mathbf{cc} \rrbracket$. Given a clock valuation \mathbf{c} and $\delta \in \mathbb{R}_+$, by $\mathbf{c} + \delta$ we denote a clock valuation \mathbf{c}' such that $\mathbf{c}'(x) = \mathbf{c}(x) + \delta$ for all $x \in \mathcal{X}$. Moreover, for a clock valuation \mathbf{c} and an assignment $\mathbf{a} \in \text{Asg}(\mathcal{X})$, by $\mathbf{c}(\mathbf{a})$ we denote a clock valuation \mathbf{c}' such that for all $x \in \mathcal{X}$ it holds $\mathbf{c}'(x) = \mathbf{c}(\mathbf{a}(x))$ if $\mathbf{a}(x) \in \mathcal{X}$, and $\mathbf{c}'(x) = 0$ otherwise (i.e., if $\mathbf{a}(x) = x_0$). Finally, by \mathbf{c}^0 we denote the *initial* clock valuation, i.e., the valuation such that $\mathbf{c}^0(x) = 0$ for all $x \in \mathcal{X}$.

Definition 11. A timed automaton with discrete data and parametric assignments (*TADDPA*) is a tuple $\mathcal{A} = (\mathcal{L}, L, l^0, DV, DP, \mathcal{X}, \mathcal{E}, \mathcal{I}_c, \mathcal{I}_v, \mathbf{v}^0)$, where \mathcal{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 \mathcal{T}), DP is a finite set of parameters (of the types in \mathcal{T}), \mathcal{X} is a finite set of clocks, $\mathcal{E} \subseteq L \times L \times \text{BoE}(DV) \times \mathcal{C}_{\mathcal{X}}(\mathcal{X}) \times \text{InS}^{\diamond}(DV, DP) \times \text{Asg}(\mathcal{X}) \times L$ is a transition relation, $\mathcal{I}_c : L \rightarrow \mathcal{C}_{\mathcal{X}}(\mathcal{X})$ and $\mathcal{I}_v : L \rightarrow \text{BoE}(DV)$ are, respectively a clocks' and a variables' invariant functions, and $\mathbf{v}^0 : DV \rightarrow \mathcal{T}$ s.t. $\mathbf{v}^0 \models \mathcal{I}_v(l^0)$ is an initial variables valuation.

The invariant functions assign to each location a clock constraint and a boolean expression specifying the conditions under which \mathcal{A} can stay in this location. Each element $t = (l, \mathfrak{l}, \beta, \mathbf{cc}, \alpha, \mathbf{a}, l')$ $\in \mathcal{E}$ denotes a transition from the location l to the location l' , where \mathfrak{l} is the label of the transition t , β and \mathbf{cc} define the enabling conditions for t , α is the instruction to be performed, and \mathbf{a} is the clock assignment. Moreover, for a transition $t = (l, \mathfrak{l}, \beta, \mathbf{cc}, \alpha, \mathbf{a}, l') \in \mathcal{E}$ we write *source*(t), *label*(t), *vguard*(t), *cguard*(t), *instr*(t), *asgn*(t) and *target*(t) for l , \mathfrak{l} , β , \mathbf{cc} , α , \mathbf{a} and l' respectively.

Semantics of the above automata is given as follows:

Definition 12. Semantics of a TADDPA $\mathcal{A} = (\mathcal{L}, L, l^0, DV, DP, \mathcal{X}, \mathcal{E}, \mathcal{I}_c, \mathcal{I}_v, \mathbf{v}^0)$ for a parameter valuation $\mathbf{p} : DP \rightarrow \mathcal{T}$ is a labelled transition system¹⁰ $\mathcal{S}(\mathcal{A}, \mathbf{p}) = (Q, q^0, \mathcal{L}_{\mathcal{S}}, \rightarrow)$, where:

¹⁰ By a labelled transition system we mean a tuple $\mathcal{S} = (S, s^0, A, \rightarrow)$, where S is a set of states, $s^0 \in S$ is the initial state, A is a set of labels, and $\rightarrow \subseteq S \times A \times S$ is a (labelled) transition relation.

- $Q = \{(l, \mathbf{v}, \mathbf{c}) \mid l \in L \wedge \forall v \in DV \mathbf{v}(v) \in \text{type}(v) \wedge \mathbf{c} \in \mathbb{R}_+^{|\mathcal{X}|} \wedge \mathbf{c} \models \mathcal{I}_c(l) \wedge \mathbf{v} \models \mathcal{I}_v(l)\}$ is the set of states,
- $q^0 = (l^0, \mathbf{v}^0, \mathbf{c}^0)$ is the initial state,
- $\mathcal{L}_S = \mathcal{L} \cup \mathbb{R}_+$ is the set of labels,
- $\rightarrow \subseteq Q \times \mathcal{L}_S \times Q$ is the smallest transition relation defined by the rules:
 - for $\iota \in \mathcal{L}$, $(l, \mathbf{v}, \mathbf{c}) \xrightarrow{\iota} (l', \mathbf{v}', \mathbf{c}')$ iff there exists a transition $t = (l, \iota, \beta, \mathbf{cc}, \alpha, \mathbf{a}, l') \in \mathcal{E}$ such that $\mathbf{v} \models \mathcal{I}_v(l)$, $\mathbf{c} \models \mathcal{I}_c(l)$, $\mathbf{v} \models \beta$, $\mathbf{c} \models \mathbf{cc}$, $\mathbf{v}' = \mathbf{v}(\alpha, \mathbf{p}) \models \mathcal{I}_v(l')$, and $\mathbf{c}' = \mathbf{c}(\mathbf{a}) \models \mathcal{I}_c(l')$ (action transition),
 - for $\delta \in \mathbb{R}_+$, $(l, \mathbf{v}, \mathbf{c}) \xrightarrow{\delta} (l, \mathbf{v}, \mathbf{c} + \delta)$ iff $\mathbf{c}, \mathbf{c} + \delta \models \mathcal{I}_c(l)$ (time transition).

A transition $t \in \mathcal{E}$ is *enabled* at a state $(l, \mathbf{v}, \mathbf{c})$ for a given parameter valuation \mathbf{p} if $\mathbf{v} \models \text{vguard}(t)$, $\mathbf{c} \models \text{cguard}(t)$, $\mathbf{c}(\text{asgn}(t)) \models \mathcal{I}_c(\text{target}(t))$, and $\mathbf{v}(\text{instr}(t), \mathbf{p}) \models \mathcal{I}_v(\text{target}(t))$. 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 $q = (l, \mathbf{v}, \mathbf{c})$ the system can either execute an enabled transition t and move to the state $q' = (l', \mathbf{v}', \mathbf{c}')$ where $l' = \text{target}(t)$, the valuation of variables is changed according to $\text{instr}(t)$ and the parameter valuation \mathbf{p} , and the clock valuation is changed according to $\text{asgn}(t)$, or move to the state $q' = (l, \mathbf{v}, \mathbf{c} + \delta)$ which results from passing some time $\delta \in \mathbb{R}_+$ such that $\mathbf{c} + \delta \models \text{inv}(l)$.

We say that a location l (a variables valuation \mathbf{v} , respectively) is *reachable* if some state (l, \cdot, \cdot) ($(\cdot, \mathbf{v}, \cdot)$, respectively) is reachable in $\mathcal{S}(\mathcal{A}, \mathbf{p})$. Given $D \subseteq DV$, a partial variables valuation $\mathbf{v}_D : D \rightarrow \mathcal{T}$ is reachable if some state $(\cdot, \mathbf{v}, \cdot)$ s.t. $\mathbf{v}|_D = \mathbf{v}_D$ is reachable in $\mathcal{S}(\mathcal{A}, \mathbf{p})$.

5.2 SAT-Based Reachability Checking

In the paper [19] we showed how to test reachability for timed automata with discrete data (TADD) using SAT-based bounded model checking to this aim. The main idea consisted in discretising the set of clock valuation of the automaton considered, in order to obtain a countable state space. Next, the transition relation of the transition system obtained was unfolded up to some depth k , and the unfolding was encoded as a propositional formula. The property to be tested was encoded as a propositional formula as well, and satisfiability of the conjunction of these two formulas was checked using a SAT-solver. Satisfiability of the conjunction allowed to conclude that a path from the initial state to a state satisfying the property was found.

Comparing with the automata considered in [19], the automata used in this paper are extended in the following way:

- values of discrete variables are not only integers, but are of several discrete types,
- arithmetic expressions used can be of a more involved 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 instructions can be 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 [19]. The way of extending arithmetic operations on integers was described in [18]. 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 γ 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 $\pi = w_0 \rightarrow w_1 \rightarrow \dots \rightarrow w_n$ ($n \in \mathbb{N}$) in the abstract graph, such that $w_0 \in V_p$ (i.e., is an initial world) and $w_n \in V_k$ (i.e., is a final world) - i.e., a sequence of worlds and abstract services which transform them. Let O_π be the set of all the objects which occur in all the worlds along this path (i.e., $O_\pi = \{o \in w_i \mid i = 0, \dots, n\}$). Then, we define $V(\pi) = \{objectName.attributeName \mid objectName \in O_\pi\}$, $V_{pre}(\pi) = \{objectName.attributeName.pre \mid objectName \in O_\pi \wedge \exists_{i \in \{0, \dots, n-1\}} objectName \in w_i \cap w_{i+1}\}$ 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 discrete variables of the automaton $\mathcal{A}(\pi)$ corresponding to π is equal to $V'(\pi)$. The intuition behind this construction is that for each attribute of each objects occurring along the path we define a variable aimed at storing the value of the attribute ($objectName.attributeName$). Moreover, for each such variable we introduce three new boolean variables: the one saying whether the flag `isConst` for the attribute has been set ($objectName.attributeName.isConst$), the second one to express that the attribute has been set (has a nonempty value; $objectName.attributeName.isSet$, and the third one to specify that the value of the attribute is nonempty but its exact value is not given ($objectName.attributeName.isAny$). The variables in $V_{pre}(\pi)$ (of the form $objectName.attributeName.pre$) are aimed at storing values of attributes from a pre-world of a service.

The initial values of 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 γ in w_0 , we set $x.y := \gamma$, $x.y.isAny := false$ and $x.y.isSet := true$; concerning $x.y.isConst$ we set it *true* if such a condition occurs in the query, otherwise it is set to *false*,
- for each attribute $x.y$ which according to the query is set, but its value is not given directly, we set $x.y.isSet := true$, and $x.y.isAny = true$; $x.y.isConst$ 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 $type(x.y)$),
- for each attribute $x.y$ which does not occur in the query or is specified there as having the empty value we set $x.y.isSet = false$, $x.y.isAny = true$, $x.y.isConst = false$, the value of $x.y$ is set to an arbitrary value as above,
 - each variable of the form $x.y.pre$ is assumed to have a “zero” value of $type(x.y)$.

Define for each $w_i, i = 0, \dots, n$, a new location of $\mathcal{A}(\pi)$, denoted for simplicity w_i as well, and consider an edge $w_i \rightarrow w_{i+1}$ of π ($i \in \{0, \dots, n-1\}$), corresponding to an abstract service sa_i . For each concrete service s of the type of sa_i we introduce a new location w_i^s and the transitions $w_i \xrightarrow{s} w_i^s$ and $w_i^s \xrightarrow{\varepsilon} w_{i+1}$ (where ε is an “empty” label)¹¹. Then, we make use of the description of s as follows:

- the precondition of s becomes the guard of the transition $w_i \xrightarrow{s} w_i^s$ (notice that a disjunctive form is here allowed);
- the list `requires` of s is used to construct the instruction α “decorating” $w_i \xrightarrow{s} w_i^s$: initially α is set to ε , then, for each attribute y of an object x occurring in `requires` for which it holds $x.y.isSet = true$, α is extended by concatenating $x.y.pre := x.y$,
- the lists `mustSet` and `mustSetConst` of s are used to construct the instruction α as well: for each attribute $x.y$ occurring in the list `mustSet` α is extended by concatenating $x.y.isSet := true$, and for each attribute $x.y$ occurring in the list `mustSetConst` of s α 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 α “decorating” $w_i \xrightarrow{s} w_i^s$ by concatenating respectively $x.y.isSet := true$ or $x.y.isConst := true$ if such an instruction has not been added to α before (or respectively $x.y.isSet := false$ or $x.y.isConst := false$ if the predicates are negated)¹²,
 - each predicate of the form $x.y = z$ or `post(x.y)=z` (where z can be either a concrete value or an expression¹³) results in extending the instruction α 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 `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 α 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.

¹¹ We assume here that the postcondition of s contains no disjunctions; otherwise we treat s a number of concrete services each of which has the postcondition corresponding to one part of the DNF in the original postcondition of s .

¹² 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.

¹³ Recall that the expressions can refer only to values the variables have in the pre-world of a service.

- moreover, for each attribute $x.y$ which occurs either in `mustSet` or in the postcondition in a predicate `isSet(x.y)`, but does not have in the `postCondition` any “corresponding” predicate which allows to set its value, we introduce a new parameter $p_{x.y}^s$, and extend α 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 ε are set to *true*. The set of instructions of the latter transition is empty. The set of clocks of $\mathcal{A}(\pi)$ 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, 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 `requires` their previous values are stored.

The above construction can be optimised in several ways. Firstly, one can add a new “intermediate” location w_i^s 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 a parameter to be assigned to a variable $x.y$, then we can reduce the number of parameters: instead of introducing a new parameter for each concrete service we can introduce one parameter $p_{x.y}^{t,i}$. This follows from the fact that only one concrete service of this type is executed as the i -th, and therefore only one assignment a new value to $x.y$ is performed.

6 Experimental Results and Concluding Remarks

The method described above has been implemented. The preliminary implementation was tested on a Getting Juice example considered in [3], by running it to generate a sequence of concrete services corresponding to the abstract path `SelectWare`, then `FruitSelling` and then `MakingJuice`. The sequence has been found; a detailed description of the example together with the result can be found in the appendix.

Currently, the automaton is generated by hand, since a repository of concrete services is still under construction. In the future we are going to automate the method completely, including dynamic translation of a service, dynamic creation of enumeration types based on the query and on the contents of the repository, and building the automaton step by step. This will enable us to test efficiency of the approach.

References

1. S. Ambroszkiewicz. *enTish: An Approach to service Description and Composition*. ICS PAS, Ordona 21, 01-237 Warsaw, 2003.
2. DAML-S (and OWL-S) 0.9 draft release. <http://www.daml.org/services/daml-s/0.9/>, 2003.
3. M. Jarocki, A. Niewiadomski, W. Penczek, A. Pótróla, and M. Szreter. Towards automatic composition of web services: Abstract planning phase. Technical Report 1017, ICS PAS, Ordona 21, 01-237 Warsaw, February 2010.
4. M. Klusch, A. Geber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In *Proc. of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*. AAAI Press, 2005.
5. D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the Planning Domain Definition Language - version 1.2. Technical Report TR-98-003, Yale Center for Computational Vision and Control, 1998.
6. S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proc. of the 11st Int. World Wide Web Conference (WWW'02)*, 2002.
7. J. Rao. *Semantic Web Service Composition via Logic-Based Program Synthesis*. PhD thesis, Dept. of Comp. and Inf. Sci., Norwegian University of Science and Technology, 2004.
8. J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process model. In *Proc. of the IEEE Int. Conf. on Web Services (ICWS'04)*. IEEE Computer Society, 2004.
9. J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, pages 43–54, 2004.
10. D. Redavid, L. Iannone, and T. Payne. OWL-S atomic services composition with SWRL rules. In *Proc. of the 4th Italian Semantic Web Workshop: Semantic Web Applications and Perspectives (SWAP 2007)*, 2007.
11. RSat. <http://reasoning.cs.ucla.edu/rsat>, 2006.
12. E. Sirin, J. Hendler, and B. Parsia. Semi-automatic compositions of web services using semantic description. In *Proc. of the Int. Workshop 'Web Services: Modeling, Architecture and Infrastructure' (at ICEIS 2003)*, 2003.
13. SOAP version 1.2. <http://www.w3.org/TR/soap>, 2007.
14. B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *Proc. of Int. Workshop on Planning for Web Services (at ICAPS 2003)*, 2003.
15. Universal Description, Discovery and Integration v3.0.2 (UDDI). <http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm>, 2005.
16. Web Services Business Process Execution Language v2.0. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
17. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>, 2001.
18. A. Zbrzezny. A boolean encoding of arithmetic operations. Technical Report 999, ICS PAS, 2007.
19. A. Zbrzezny and A. Pótróla. SAT-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 70(1-2):579–593, 2007.

A Experimental Results - A Detailed Description

Below we present the Getting Juice example considered in [3]. Assume we have the following classes:


```

Ware      id      integer
Ware      name    string
Ware      owner   string
Measurable capacity float
Juice     extends Ware, Measurable
Fruits    extends Ware, Measurable

```

and the following types of services:

```

SelectWare produces    w:Ware
SelectWare consumes    null
SelectWare requires    null
SelectWare mustSet     w.name; w.owner

Selling produces       null
Selling consumes       null
Selling requires       w:Ware
Selling mustSet        w.id; w.owner
Selling precondition   not isSet(w.id) and isSet(w.name)
                        and isSet(w.owner)
Selling postCondition  w.owner!=pre(w).owner

FruitSelling extends   Selling
FruitSelling requires  w:Fruits
FruitSelling mustSet   w.capacity
FruitSelling postCondition w.capacity>0

JuiceSelling extends   Selling
JuiceSelling requires  w:Juice
JuiceSelling mustSet   w.capacity
JuiceSelling postCondition w.capacity>0

MakingJuice produces   j:Juice
MakingJuice consumes   f:Fruits
MakingJuice mustSet    j.id; j.name; j.capacity
MakingJuice precondition isSet(f.id) and isSet(f.name) and
                        isSet(f.owner) and f.capacity>0
MakingJuice postCondition isSet(j.id) and isSet(j.name) and
                        j.capacity>0

```

The user's query is specified as follows:

```

InitWorld    null
InitClause   true
EffectWorld  j:Juice
EffectClause j.id>0 and j.capacity=10 and j.owner="Me"

```

One of the sequences of services which possibly can lead to satisfying the query is `SelectWare`, then `FruitSelling` and then `MakingJuice` [3]. Below we consider concretising the above path of the abstract graph.

Assume the concrete instances of the `SelectWare` specify the following offers¹⁴:

¹⁴ The current version of our implementation does not deal with inheritance of classes.

```

FruitNetMarket mustSet      w.name, w.owner
FruitNetMarket preCondition  -
FruitNetMarket postCondition (w.name=strawberry and
                             w.owner=shop1) or
                             (w.name=blueberry and w.owner=shop1)

FruitNetOffers mustSet      w.name, w.owner
FruitNetOffers preCondition  -
FruitNetOffers postCondition (w.name=plum and w.owner=shop2) or
                             (w.name=apple and w.owner=shop2) or
                             (w.name=apple and w.owner=shop3)

```

Next, the fruitselling services specify:

```

Shop1 mustSet      w.id, w.owner, w.capacity
Shop1 precondition not isSet(w.id) and isSet(w.name)
                  and isSet(w.owner) and w.owner=shop1
Shop1 postcondition w.owner!=pre(w).owner and w.id>0 and
                  w.capacity>0 and w.capacity<=10

Shop2 mustSet      w.id, w.owner, w.capacity
Shop2 precondition not isSet(w.id) and isSet(w.name)
                  and isSet(w.owner) and w.owner=shop2
Shop2 postcondition w.owner!=pre(w).owner and w.id>0 and
                  w.capacity>0

Shop3 mustSet      w.id, w.owner, w.capacity
Shop3 precondition not isSet(w.id) and isSet(w.name)
                  and isSet(w.owner) and w.owner=shop3
Shop3 postcondition w.owner!=pre(w).owner and w.id>0
                  and w.capacity>=100

```

which means that Shop1 is able to sell at most 10 units of fruits, Shop2 - at least 100 units, while Shop3 is able to sell any amount. Finally, we have the following services which make juice:

```

HomeJuiceMaking mustSet      j.id, j.name, j.capacity
HomeJuiceMaking preCondition isSet(f.id) and isSet(f.name) and
                             isSet(f.capacity) and isSet(f.owner)
                             and f.capacity>0
                             and f.capacity<=10 and
                             f.name!=plum and f.name!=apple
HomeJuiceMaking postCondition isSet(j.id) and isSet(j.name) and
                             j.capacity>0 and j.name=f.name
                             and j.capacity=f.capacity
                             and j.owner=f.owner

GrandmaKitchen mustSet      j.id, j.name, j.capacity
GrandmaKitchen preCondition isSet(f.id) and isSet(f.name) and
                             isSet(f.capacity) and isSet(f.owner)
                             and f.capacity>0 and f.capacity<=5

```

```

GrandmaKitchen postCondition isSet(j.id) and isSet(j.name) and
                               j.capacity>0 and j.name=f.name
                               and j.capacity=f.capacity
                               and j.owner=f.owner

JuiceTex      mustSet      j.id, j.name, j.capacity
JuiceTex      preCondition  isSet(f.id) and isSet(f.name) and
                               isSet(f.capacity) and isSet(f.owner)
                               and f.capacity>0

JuiceTex      postCondition isSet(j.id) and isSet(j.name) and
                               j.capacity>0 and j.name=f.name
                               and j.capacity=2*f.capacity
                               and j.owner=f.owner

```

Thus, 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 ranges of enumeration types can be deduced from the offers and from the user's query). The variables and their types are: $f.id : integer$, $f.name : FruitTypes$, $f.owner : OwnerNames$, $f.capacity : float$, $j.id : integer$, $j.name : FruitTypes$, $j.owner : OwnerNames$, $j.capacity : float$ plus the corresponding boolean variables of the form $x.y.isSet$, $x.y.isAny$ and $x.y.isConst$. Moreover, we introduce one additional variable $f.owner.pre : OwnerNames$ to store the previous value of $f.owner$ ¹⁵.

All the variables of the form x,y are initialised to zero values of the appropriate types, each $x.y.isSet$ and $x.y.isConst$ is initialised with *false*, and each $x.y.isAny$ is initialised with *true*.

- The `FruitNetMarket` generates two transitions (together with the intermediate locations and the “ ε -transitions” outgoing them). The first one is decorated with $f.name.isSet := true; f.owner.isSet := true; f.name := strawberry; f.owner := shop1; f.owner.isAny := false; f.name.isAny := false$, the second one is decorated in a similar way, but with $f.name := blueberry$, the edges for the three offers of `FruitNetOffers` look similarly,
- the fruitselling services correspond to the following edges and intermediate locations:
 - for Shop1:
 - * the guard of the first edge is $f.id.isSet = false \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.owner = shop1$,
 - * the instruction is $f.id.isSet := true; f.owner.isSet := true; f.capacity.isSet := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p_{f.owner}^{FS}; f.id := p_{f.id}^{FS}; f.capacity := p_{f.cap}^{FS}$ (where p^{FS} are parameters),
 - * the invariant of the intermediate location is $\neg(f.owner.pre = f.owner) \wedge f.id > 0 \wedge f.capacity > 0 \wedge f.capacity \leq 10$;

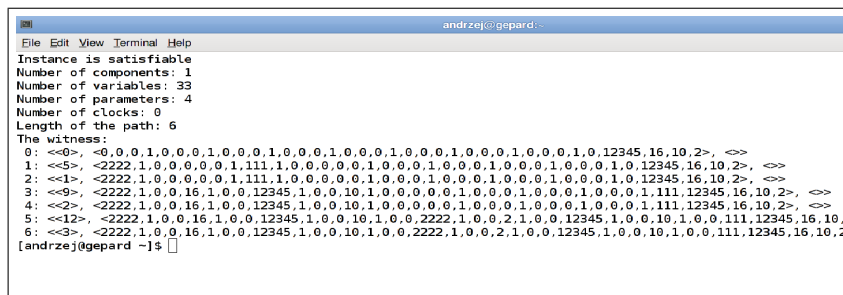
¹⁵ We apply the optimisation allowing to add one variable of the form $x.y.pre$ only, as well as the one consisting in reducing the number of parameters.

- for Shop2:
 - * the guard of the first edge is $f.id.isSet = false \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.owner = shop2$,
 - * the instruction is $f.id.isSet := true; f.owner.isSet := true; f.capacity.isSet := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p_{f.own}^{FS}; f.id := p_{f.id}^{FS}; f.capacity := p_{f.cap}^{FS}$,
 - * the invariant of the intermediate location is $\neg(f.owner.pre = f.owner) \wedge f.id > 0 \wedge f.capacity > 0$,
- for Shop3:
 - * the guard of the first edge is $f.id.isSet = false \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.owner = shop3$,
 - * the instruction is $f.id.isSet := true; f.owner.isSet := true; f.capacity.isSet := true; f.id.isAny := false; f.owner.isAny := false; f.capacity.isAny := false; f.owner.pre := f.owner; f.owner := p_{f.own}^{FS}; f.id := p_{f.id}^{FS}; f.capacity := p_{f.cap}^{FS}$,
 - * the invariant of the intermediate location is $\neg(f.owner.pre = f.owner) \wedge f.id > 0 \wedge f.capacity \geq 0$,
- for the services making juice from fruits:
 - for HomeJuiceMaking:
 - * the guard of the first edge is $f.id.isSet = true \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.capacity.isSet = true \wedge f.capacity > 0 \wedge f.capacity \leq 10 \wedge \neg(f.name = plum) \wedge \neg(f.name = apple)$
 - * the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.capacity.isSet := true; j.capacity.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p_{j.id}^{MJ}; j.capacity := f.capacity; j.name := f.name; j.owner := f.owner$
 - * the invariant of the intermediate location is $j.capacity > 0$
 - for GrandmaKitchen
 - * the guard of the first edge is $f.id.isSet = true \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.capacity.isSet = true \wedge f.capacity > 0 \wedge f.capacity \leq 5$,
 - * the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.capacity.isSet := true; j.capacity.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p_{j.id}^{MJ}; j.capacity := f.capacity; j.name := f.name; j.owner := f.owner$
 - * the invariant of the intermediate location is $j.capacity > 0$
 - for JuiceTex
 - * the guard of the first edge is $f.id.isSet = true \wedge f.name.isSet = true \wedge f.owner.isSet = true \wedge f.capacity.isSet = true \wedge f.capacity > 0$
 - * the instruction is $j.id.isSet := true; j.id.isAny := false; j.name.isSet := true; j.name.isAny := false; j.capacity.isSet := true; j.capacity.isAny := false; j.owner.isSet := true; j.owner.isAny := false; j.id := p_{j.id}^{MJ}; j.capacity := 2 * f.capacity; j.name := f.name; j.owner := f.owner$

* the invariant of the intermediate location is $j.capacity > 0$

The condition to be tested is $j.id.isSet \wedge \neg j.id.isAny \wedge j.capacity.isSet \wedge \neg j.capacity.isAny \wedge j.owner.isSet \wedge \neg j.owner.isAny \wedge j.id > 0 \wedge j.capacity = 10 \wedge j.owner = me$. In practice, we extend it by adding an additional proposition which is true in the locations w_0, \dots, 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. A screenshot displaying the witness is presented in Fig. 1. To find the



```

andrze@gepard:~$
File Edit View Terminal Help
Instance is satisfiable
Number of components: 1
Number of variables: 33
Number of parameters: 4
Number of clocks: 0
Length of the path: 6
The witness:
0: <<0>, <0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,12345,16,10,2>, <<>
1: <<5>, <2222,1,0,0,0,0,0,1,111,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,12345,16,10,2>, <<>
2: <<1>, <2222,1,0,0,0,0,0,1,111,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,12345,16,10,2>, <<>
3: <<9>, <2222,1,0,0,16,1,0,0,12345,1,0,0,10,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,111,12345,16,10,2>, <<>
4: <<2>, <2222,1,0,0,16,1,0,0,12345,1,0,0,10,1,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,111,12345,16,10,2>, <<>
5: <<12>, <2222,1,0,0,16,1,0,0,12345,1,0,0,10,1,0,0,2222,1,0,0,2,1,0,0,12345,1,0,0,10,1,0,0,111,12345,16,10,2>, <<>
6: <<3>, <2222,1,0,0,16,1,0,0,12345,1,0,0,10,1,0,0,2222,1,0,0,2,1,0,0,12345,1,0,0,10,1,0,0,111,12345,16,10,2>, <<>
[andrze@gepard ~]$

```

Fig. 1. A witness for concretisation of an abstract path for the Getting Juice example

witness, we checked satisfaction of the boolean formula encoding the translation of the tested condition. The formula in question consisted of 20152 variables and 52885 clauses; our implementation needed 0.65 second and 6.2 MB memory to produce it. Its satisfiability was checked by RSAT[11], a mainstream SAT solver; to checking it 0.1 seconds and 5.6 MB of memory were needed.