PRACE IPI PAN ICS PAS REPORTS

Maciej Szreter

Bounded Abstract Planning in PlanICS based on Graph Databases

Nr 1031

INSTYTUT PODSTAW INFORMATYKI POLSKIEJ AKADEMII NAUK
INSTITUTE OF COMPUTER SCIENCE POLISH ACADEMY OF SCIENCES
ul. Jana Kazimierza 5, 01-248 Warszawa http://www.ipipan.waw.pl

Maciej Szreter

Bounded Abstract Planning in PlanICS based on Graph Databases

Nr 1031

Warsaw, December 2015

Pracę zgłosił Prof. dr hab. inż. Wojciech Penczek

Adres autora: Instytut Podstaw Informatyki PAN
ul. Jana Kazimierza 5
01-248 Warszawa
Polska
E-mail: mszreter@ipipan.waw.pl

Symbol klasyfikacji rzeczowej MSC 2000: 03D99, 68Q05

Na prawach rękopisu
Printed as manuscript

Nakład 80 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w grudniu 2015 r. Wydawnictwo IPI PAN. ISSN 0138-0648

Abstract

The paper describes an application of a graph database to the abstract planning in the PlanICS composition system for web services. Abstract planning is the first stage of the service composition process, and consists in matching types of the services and objects processed by them, with some additional constraints. The result is an abstract plan matching the user query. The presented solution prunes the ontologies for the abstract planners, greatly improving efficiency and providing better scalability. This is of particular importance in the domain of web service composition, because usually systems are expected to produce answers immediatly.

Keywords: automatic composition of web services, graph databases, abstract planning

Streszczenie

Ograniczone abstrakcyjne planowanie w systemie PlanICS wykorzystujące grafowe bazy danych

Praca opisuje zastosowanie grafowej bazy danych do fazy planowania abstrakcyjnego w systemie automatycznej kompozycji usług sieciowych PlanICS. Planowanie abstrakcyjne jest pierwszą fazą procesu planowania usług, i polega na dopasowaniu typów usług i przetwarzanych przez nie obiektów, w celu wygenerowania planu abstrakcyjnego spełniającego zapytanie skierowane do systemu przez użytkownika. Rozwiązanie bardzo istotnie zwiększa efektywność istniejących metod planowania opartych na testowaniu spełnialności formuł logicznych lub bazujących na algorytmach genetycznych. Jest to szczególnie ważne w dziedzinie planowania usług sieciowych, gdzie od systemów oczekuje się bardzo krótkich czasów przetwarzania zapytań. Metoda ogranicza przeszukiwaną ontologię ze względu na zapytanie użytkownika, znacząco ułatwiając znalezienie rozwiązań.

Słowa kluczowe: automatyczna kompozycja usług sieciowych, grafowe bazy danych, planowanie abstrakcyjne

1 Introduction

PlanICS [16] is a framework for automated composition of web services. Its distinguishing features are precise definition of the semantics, a strong type system for service and object types kept in an ontology, and two-stage composition. First (abstract) stage dealing with types, while second (concrete) stage works with instances of services, called offers, and objects processed by services (referred to as objects). The aim of PlanICS is to integrate services designed in different formalisms, either defined as 'black-boxes' or ascribed with semantic descriptions. Another feature is the ability to deal with big numbers of services, partly enforced by the idea of abstract planning and partly by the implementations of planners, inspired by model checking and soft computing areas with handling huge state spaces as an objective.

Abstract planners are tools performing the abstract planning stage. The planners developed so far are based on Satisfiability-Modulo-Theorems (SMT) solvers, and genetic algorithms. The former suffer from the combinatorial explosion, especially for longer plans, and the latter are fast but in most cases incomplete. This paper sets out to prove that a pre-planner based on (graph) databases offers several advantages. The major profit is that it usually prunes the ontology significantly. In contrast, all the existing solvers encode whole ontologies and then use heuristics to restrict the search to the relevant fragment. When modeling the problems as graphs, this fragment is determined by a graph database. The abstract planning problem has the structure well suited towards applying a database: the modifications of the ontology are relatively rare, and big numbers of queries are expected to be processed quickly.

Graph databases are a recent development among several approaches to redefine traditional relational database model, referred to by a common name of *NoSQL* [17]. These approaches use different data models as a background. In the case of graph databases, data are represented as graphs, and the database engine provides efficient representation, as well as traversal and search methods, based on a specialized query languages. Neo4j [11] is one of the most popular graph databases. A part of Neo4j is *Cypher* language for expressing queries, which in general correspond to reachability problems for graphs, with several additional constraints.

The general idea of our approach is to model the relations of processing objects by services as a graph. Then, the problem of reachability between nodes modeling objects from the input and expected worlds of the user query corresponds to potential context partial orders of resulting abstract plans. A graph database finds a subgraph relevant for the user query from the possibly huge graph representing the ontology. Finally, the search for exact abstract plans is performed by an external abstract planner, and it is in general easier because services and objects not relevant for the query

have been pruned. Using external solvers is motivated by the effectiveness and simplicity of implementation. Implementing the whole planning process in the graph database will be the part of the future work.

An additional advantage of the presented approach are nice visualization capabilities provided by graph databases, allowing human operators to easily look at the internal structure of an ontology. This may enable interesting ways of interacting with users, performing analyses which services and objects are the most intensively used etc.

The paper is structured as follows. First we report the related work. Section 2 describes the PlanICS approach to abstract planning. Section 3 introduces graphs, and then delivers the main result. The planning problem is modeled as a graph, and a graph database is applied to pruning the ontology according to the user query. Section 4 explains in detail the structure of a selected benchmark. Section 5 provides experimental results comparing the performance of the SMT-based abstract planner for full-size ontologies and those pruned by the graph database search. Section 6 concludes the paper and presents possible directions of the future work.

1.1 Related work

The presentation of the related work is centered around PlanICS, because it is where the abstract planning is defined. To the best of our knowledge, no other approach to the web service composition distinguishes abstract planning as a separate stage.

The idea of abstract and concrete planing has been inherited by PlanICS from Entish [2] language and associated composition system. A general description of PlanICS can be found in [5]. The paper [15] describes how the planners based on SMT and genetic algorithm can interact to exploit advantages of every approach. [16] confirms that custom PlanICS concrete planning algorithms are indeed much more effective than available general approaches which can be applied to this aim. [12] extends the basic planning scheme into temporal planning, allowing for more expressible user queries, referring to several worlds and relations between them. In [8] the authors perform a static analysis determining which actions are certain not to produce any plan, and feed this information to an SMT solver, adding it to the result formula as a set of constraints. The experiments show that some improvement is gained at the stage of searching for solutions, and checking that no more solution exist takes longer time. Our method is different by performing all the abstract planning (without checking pre/post constraints) by explicit graph algorithms, thus pruning the ontology. All the PlanICS abstract planners (to the date based on SMT or genetic algorithms) can be used for checking it

Concerning the other automatic approaches to web services composition, [21] is a solution the closest to ours, as it defines the services in a similar way,

based on pre- and postconditions. It also models ontology using an object concept. Compared to PlanICS, it is focused more on logical deductions based on queries, and less on efficiency (has no two-phase planning).

There are several papers solving the web composition problem by modeling it in the domain of graphs, with solutions corresponding to reachability in graphs. In these papers, services are based on the IOPR (input,output,precondition,result) paradigm, similarly to our approach. None of the papers distinguishes between abstract and concrete planning in our sense, and neither uses graph databases. To the best of our knowledge, we could also find no experimental analysis showing that graph-based approaches can effectively deal with ontologies with significant numbers of services and objects. In [7], information about inputs and outputs of services are represented by interface automata. In dependency graphs, nodes correspond to object types and edges to services processing them. There is no experimental evaluation. [1] represents both services and objects processed by them by graph nodes, and uses backward chaining for searching plans satisfying the user query. The paper is a sketch rather than a complete solution. In [10], nodes correspond to web services, but there is no abstract phase and no objects directly represented in the graph. Edges are added at the basis of concrete values of the parameters. There are no experimental results and the authors seem not to care much about the efficiency of the method. [6] uses weighted graphs for modeling parameters such as cost, execution time and availability, and presents respective graph algorithms for solving the corresponding composition problem. [9] describes a graph approach to composing web services. It does not use a graph database, and deals not with the abstract planning of any kind.

Preprocessing and pruning the state space using graph algorithms has been examined in [4], where a preliminary over-approximation graph is generated in order to perform the exact planning on it.

[20] annotates ontologies with semantics information, and uses on top of this a graph algorithm with similarity measures describing how services are matched.

Graph databases [18, 3] are a relatively recent addition in the domain of NoSQL databases, defined by rejecting the traditional database model of relational tables, and using alternative solutions, in this case graphs. Neo4j is one of the most popular implementations. [19] performs web service composition using the MapReduce approach aimed at processing Big Data. Our method uses a different solution, but shares the idea of exploiting efficient tools developed for dealing with big amounts of data.

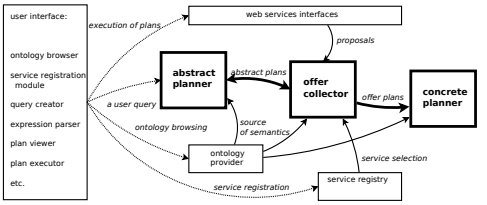

Figure 1: PlanICS overview. The rectangles stand for software components of the system. The bold arrows correspond to computation of a plan, the thin arrows model the planner infrastructure, the dotted arrows represent the user interaction.

2 Abstract planning overview

PlanICS is a system implementing an original approach which solves the service composition problem in some clearly separated stages. Fig. 1 shows the system overview. The formalism behind PlanICS is defined very precisely. We describe it in a general way, formalizing only the parts relevant to the described work, and pointing at [13] and [14] for the complete references.

2.1 Basic syntax and semantics of objects and services

The basic building blocks of PlanICS are objects and (web) services. The services read and transform the objects, and the objects contain attributes, being either other objects, or simple types such as real or integer numbers, strings, etc. We denote by $\mathbb{I}$ the set of all identifiers, and by $\mathbb{A}$ the set of all the attributes, with $\mathbb{A} \subset \mathbb{I}$.

An *object type* is a pair $(t, Attr)$, where $t \in \mathbb{I}$ and $Attr \subseteq \mathbb{A}$. $\mathbb{P}$ is the set of all object types. The objects types are composed of typed attributes with domains. An *object* is a pair $(id, type)$, where $id \in \mathbb{I}$ and $type \in \mathbb{P}$. We denote by $id : type$ assigning $type$ to the object id.

The services process objects, and have attribute lists *in*, *out*, *inout* referring to the objects read by a service (this does not mean being discarded), produced by it, or both read and written, respectively. Moreover, abstract formulas, in disjunctive normal form without negations, are defined over the attributes, built from predicates $isNull(a)$ and $isSet(a)$ stating that attribute a is not set or set. An *user query* expresses what is the input and what the output of the composition, and is technically modeled similarly to a service. For a service s or a user query q, a *specification* is $spec_x = (in_x, out_x, inout_x, pre_x, post_x)$, where $x \in \{s, q\}$, with meaning of attribute lists described above, and pre_x and $post_x$ are abstract formulas over attributes from lists.

Types of services and objects are *classes* organized in an inheritance tree rooted in the abstract type *Thing* (abstract types cannot be instantiated). All the services and objects are derived from the abstract types *Service* and

$Artifact$, respectively. The types are stored in an *ontology*. We define a transitive, irreflexive, and antisymmetric *inheritance* relation $Ext \subseteq \mathbb{P} \times \mathbb{P}$, such that $((t_1, A_1), (t_2, A_2)) \in Ext$ iff $t_1 \neq t_2$ and $A_1 \subseteq A_2$. That is, a subtype contains all the attributes of a base type and optionally introduces more attributes.

Having defined the syntax, we now move towards the PlanICS semantics. For an abstract formula α, a *valuation of object attributes* is a partial function assigning, for every object, to every its attribute a a value $true$ or $false$ if $isSet(a)$ or $isNull(a)$ predicates occur in a clause in α, respectively, or is undefined otherwise. By an *object state* we mean an object with the assigned valuation. A *world* is a pair consisting of a set of objects, and a valuation function defined for them.

From the semantic point of view, the service types and user queries are defined by their *interpretations*, which are pairs of worlds. In the first one, called *input world set* for services and *initial world set* for a query, there are objects of types belonging to *in* or *inout* lists, and the valuation is the family of the valuation functions over pre_x. In the second one, called *output world set* for services and *expected world set* for a query, there are objects from *inout* or *out* lists, and the valuation is the family of the valuation functions over $post_x$.

2.2 Abstract planning

An interpretation of a specification, as defined above, determines the pair of worlds transformed by a service or a query. In planning, we want to combine several services, modifying (parts of) consecutive worlds so that the query would be satisfied. We now present how this is done in PlanICS.

We say that two valuations for two objects are *compatible* if the types of both objects are the same, or one of them is a subtype of the another one, and the valuations agree for every attribute. Similar worlds are identified by the notion of *world compatibility*: worlds w and w' are compatible if they contain the same number of objects, and for every object state from w there exists a compatible object state from w', and vice versa. Worlds with different numbers of objects are compatible if there exist a subworld in the bigger world compatible with the smaller one.

The key idea of the abstract planning is transforming worlds by services in order to satisfy the user query. A *context function* describes a mapping of objects from the initial, expected and intermediate worlds into the attributes of user query and services. Then, for two worlds $w, w' \in W$, referred to as *world before* and *world after*, respectively, a *world transformation* for service s transforms w into w' in a context ctx, if these worlds can be mapped into the attribute list of s, and the additional requirements are met. Let IN be a ctx context image of the set in_s. The world before contains a subworld built over IN, compatible with a sub-world of some input world of the service

s, built over the objects from in_s. The state of the objects from IN is consistent with pre_s. The objects from IN and the objects not involved in the transformation do not change their states in the world after. The conditions for images of *inout* and *out* sets are given in [13].

A *transformation sequence seq* is a sequence of worlds transformed by services in a context. For a user query $q = (W^q_{init}, W^q_{exp})$, *seq* is a *user query solution* of q if there exists a world $w \in W^q_{init}$ and some world w' such that *seq* leads from w to w', and w' is compatible with some $w^q_{exp} \in W^q_{exp}$.

Finally, we want to abstract away the ordering of services where it is irrelevant. We introduce an equivalence relation of solutions: two solutions are equivalent if the number of occurrences for every service type are equal in them. For a sequence *seq*, a *context abstract plan (CAP)* is the set of all the solutions equivalent to *seq* w.r.t. the relation defined above.

2.3 Collecting offers and concrete planning

While this article covers only the abstract planning, the brief description of the concrete planning is given in order to explain how the planning process as a whole works in PlanICS.

In the second planning stage CAP is used by an *offer collector* (OC), i.e., a tool which in cooperation with the service registry queries real-world services (see Fig. 1). The service registry keeps an evidence of real-world web services, registered accordingly to the service type system from the ontology. During the registration the service provider defines a mapping between input/output data of the real-world service and the object attributes processed by the declared service type.

OC communicates with the real-world services of types present in a CAP, sending the constraints on the data, which can potentially be sent to the service in an inquiry, and on the data expected to be received in an offer in order to keep on building a potential plan. The constraints are constructed from the *pre* and *post* conditions. Usually, each service type represents a set of real-world services. Moreover, querying a single service can result in a number of offers. Thus, we define an offer set as a result of the offer collecting planning stage.

Definition 1 (Offer, offer set)**.** *Assume that the n-th instance of a service type from a CAP processes some number of objects having in total m attributes. A single offer collected by OC is a vector* $P = [v_1, v_2, \ldots, v_m]$*, where* v_j *is a value of a single object attribute from the n-th intermediate world of the CAP.*

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

The detailed translation of a PlanICS query to a set of constraints is beyond the scope of this paper, because it would require describing the whole inference process inside OC. More details on constraints can be found in [14].

Finally, the *concrete planning* consists in finding an assignment of offers to the CAP (a *concrete plan*), possibly satisfying some optimality criteria.

3 Representing Abstract Search Problem in a Graph Database

In this section we show that the abstract planning problem (restricted to matching types of services) can be translated to the reachability problem for graphs. Then we use a graph database to prune the ontology for a given user query.

3.1 Graphs

We start with introducing graphs and graph databases.

Definition 2 (Graph)**.** *A* directed graph G *is an ordered pair* (V, E) *such that* V *is a finite set of vertices and* $E \subseteq V \times V$ *is a set of ordered pairs of vertices, called edges. We write* $v_1 \rightarrow_G v_2$ *for an edge* $(v_1, v_2) \in E$*, where* $v_1, v_2 \in V$*.*

A path in a graph is a finite sequence of its vertices such that every consecutive pair of vertices is connected by an edge.

Definition 3 (Path)**.** *Given a graph* $G = (V, E)$*. A* path p *of length* k *is a finite sequence* $p = v_0, \ldots, v_k$ *of graph vertices, such that* $v_i \rightarrow_G v_{i+1}$ *for* $i \in 0, \ldots, k-1$*.*
We say that $v \in V$ *is a vertex of* p*, denoted by* $v \in p$*, if* $v = v_i$ *for some* $i \in 0, \ldots, k$*.*
Similarly, we say that $e = (v, v') \in E$ *is an edge of* p*, denoted by* $e \in p$*, if* $v = v_i$ *and* $v' = v_{i+1}$ *for some* $i \in 0, \ldots, k-1$*.*

Definition 4 (SubGraph)**.** *Given two graphs* $G = (V, E)$ *and* $G' = (V', E')$*. We say that* G' *is a* subgraph *of* G*, denoted by* $G' \subseteq G$*, if* $V' \subseteq V$ *and* $E' \subseteq E$*.*

A set of paths S_P in a graph G induces the subgraph G' such that each vertex and each transition of G' is an element of some path of S_P. This intuition is captured by the following definition:

Definition 5 (Subgraph induced by paths)**.** *Let* S_P *be a set of paths of a graph* $G = (V, E)$*. The subgraph* $G' = (V', E')$ *of* G *induced by* S_P *is defined as follows:*

- $V' = \{v \in V \mid \exists p \in S_P : v \in p\}$ *and*
- $E' = \{(v, v') \in E \mid \exists p \in S_p : (v, v') \in p\}$.

3.2 Graph databases

A *graph database* is a tool for storing directed graphs and finding their subgraphs satisfying certain properties, defined over vertices and edges that are expressed by database queries[1]. From a formal point of view, this is a graph algorithm with a precisely defined semantics. For example, one could map a group of persons to vertices, friendship relations between these persons to edges, and formulate a database query to find everyone who is female, older than 50 years (assuming that every vertex has an attribute for the age of the person it represents) and has at least two friends shared with another person.

For some applications, it is easier to consider a variant of the reachability problem of finding all the paths between two explicitly known sets of vertices: the initial and the final set. This can be easily reduced to finding all the paths between a pair of vertices, one with an outcoming edge to every element of the initial set, and the other one with an incoming edge from every vertex of the final set. The latter approach is conceptually simpler and sometimes more efficient. Since it can be applied to pruning ontologies, we describe it formally. To this aim, we begin with showing how a given graph is stored in the database, where the operations $addNode()$ and $addEdge()$, having a clear meaning, are applied.

Definition 6 (Graph database)**.** *Given a directed graph $G = (V, E)$. G is said to be* stored *in a graph database DB if $addNode(v)$ is executed for every vertex $v \in V$ and $addEdge(v, v')$ is executed for every edge $(v, v') \in E$.*

Then, searching a stored graph is defined.

Definition 7 (Graph database query result)**.** *Let $G = (V, E)$ be a directed graph stored in a graph database DB. Let $q^k_{DB} = (v_I, v_F, k)$ be a database query, where $v_I, v_F \in V$ and $k \in \mathbb{N}$. The result of the query q^k_{DB} applied to DB is the subgraph $G_q \subseteq G$, induced by the set of paths of G of length at most k, which begin with v_I and end with v_F. We refer to G_q by $result(G, q^k_{DB})$.*

From a practical perspective, graph databases can be used for storing and search effectively very big state spaces. In addition, the graph representation enables for a graphical visualization of the graphs stored in the database as well as the query results. Advanced features are available in the area of guaranteeing data redundancy, distributing data between multiple machines, and optimizing the database performance.

[1]One should not confuse a database query (specifying what is to be found in the database) and a user query (specifying the task of the composition process).

3.3 Pruning an ontology

Our graph-based approach to pruning ontologies for abstract planning consists of the following stages:

1. Choosing an upper bound (k) on the plans length for which the search is to be performed.

2. Encoding the ontology in the graph database. Every object type and service type of the ontology is represented by a distinguished vertex. The edges connect pairs of vertices, where one vertex models a service type while another one an object type. For a vertex representing a service type s, an incoming edge from a vertex representing an object type models that this object type is an input for the service type s. Similarly, an outgoing edge to a vertex representing an object type models that this object type is an output of the service type s. The rules are also applied to the object types derived from every object type occurring in the input and output lists of a service type. After application of these rules we get the *ontology graph* G_{Ont}. Later in this section we give Example 1 showing the graphs defined and used by our reduction.

3. Extending the ontology graph G_{Ont} to the *query graph* G_q, where $G_{Ont} \subseteq G_q$, by adding the *initial* vertex and the *final* vertex, for a user query q. The outgoing edges of the initial vertex are connected to the vertices representing the object types of the initial worldset of q and to the vertices representing the service types having empty input lists. The ingoing edges of the final vertex start from the vertices representing the object types of the expected worldset of q. Similarly, the corresponding edges are added for the subtypes of the object types mentioned above.

4. Searching for a set of the paths (of length restricted to k) between the initial and the final vertex of the query graph G_q. This objective is expressed by a database query fed to the graph database. The result is the *query subgraph* $G_{qs} \subseteq G_q$.

5. (Optional Postprocessing) Removing recursively from the query subgraph G_{qs} the vertices representing service types, for which some object types of their input lists have not been identified by the search. Removing them is not necessary, because we will add missing types when building the pruned ontology. However, those service types cannot be executed in the reduced ontology, so filtering them out makes it smaller and, hopefully, easier to handle for the planners. We can also remove the vertices of object types not connected to any vertices representing service types.

6. Pruning the original ontology to the service and objects types represented by the vertices of the query subgraph G_{qs}. Then, adding the types of the objects possibly produced by the service types in the reduced ontology and not relevant for any abstract plan which can be found, but required for an ontology to be complete. The pruned ontology can replace the full ontology in the planning process.

The algorithm described above can be repeated for an incremented depth k or run for any depth. It is complete and sound for the abstract plans of length restricted by the depth k chosen as a parameter. This means that the ontology pruned preserves all the abstract plans of length k and does not introduce any abstract plans which could not be generated starting with the original ontology.

In order to simplify the algorithms, we introduce two restrictions on the object and service type inheritance. 1) The multiple inheritance of objects types in not allowed. 2) Each service type is directly derived from the *Service* class.
These restrictions are technical and can be lifted at the price of making the algorithm more complex.

Now we are in a position to describe the algorithm formally.

3.3.1 Ontology Graph: Encoding an ontology

For the purpose of this section we recall some notions related to PlanICS, which are used in this section.

Definition 8 (Ontology)**.** *By an* ontology *we mean a triple* $Ont = (\mathbb{S}, \mathbb{T}, Ext)$*, where*

- $\mathbb{S}$ *is the set of all the service types,*
- $\mathbb{T}$ *is the set of all the object types, i.e., the types of* $Artifact$ *and* $Stamp$*, and their descendants,*
- Ext *is the inheritance relation of the object types.*

Moreover, we recall the function $\mathcal{T} : 2^{\mathbb{O}} \longmapsto 2^{\mathbb{T}}$, such that $\mathcal{T}(O) = \bigcup_{o \in O} \{t \in \mathbb{T} \mid t = type(o) \vee (type(o), t) \in Ext\}$ which assigns the set of the types and subtypes to each set of objects.

Assume we are given an ontology Ont. Our first task is to encode Ont as a graph. It is quite common to use graphs for modeling the inheritance of classes. We extend this approach by modeling service and object types as vertices of a graph stored in the graph database, and encoding with the edges the relation of processing and producing the objects by service types. In particular, we introduce the directed edges connecting the vertices representing the service types with the vertices corresponding to the (sub)types of the objects processed, in the way captured by the following definition.

Definition 9 (Ontology graph)**.** *Given an ontology Ont. By the* ontology graph *we mean the graph* $G_{Ont} = (V_{Ont}, E_{Ont})$*, where*

- $V_{Ont} = V_{\mathbb{S}} \cup V_{\mathbb{T}}$ *with* $V_{\mathbb{S}} = \{v_s \mid s \in \mathbb{S}\}$ *and* $V_{\mathbb{T}} = \{v_{\boldsymbol{t}} \mid \boldsymbol{t} \in \mathbb{T}\}$*,*
- $E_{Ont} = E_{\mathbb{S}} \cup E_{\mathbb{T}}$ *with*

  $E_{\mathbb{S}} = \{(v_s, v_{\boldsymbol{t}}) \mid s \in \mathbb{S} \ \wedge \boldsymbol{t} \in \mathcal{T}(out_s \cup inout_s)\}$,
  $E_{\mathbb{T}} = \{(v_{\boldsymbol{t}}, v_s) \mid s \in \mathbb{S} \ \wedge \boldsymbol{t} \in \mathcal{T}(in_s \cup inout_s)\}$.

Example 1. *In Fig. 2 the ontology pruning abstraction is applied to a simple ontology being a variant of the examples shown in the paper Plan*ICS *[16] describing Plan*ICS*. There are (among others) the object types of* $Nails$*,* $Boards$ *and* $Arbour$ *defined (all being the subtypes of* $Ware$*, and of* $Artifact$*), and there is service Selling for transactions concerning subtypes of* $Ware$*. There is service* $WoodBuilding$ *building* $Arbour$ *out of* $Nails$ *and* $Boards$*. The user query can be described as: "I have some nails and boards, I want to get an arbour".*

In the graph, the oval vertices represent the object types of the ontology, the rectangular vertices represent the service types, while the solid edges connect object types with service types according to Definition 9. The dotted edges extend the relations of processing objects by services to the inherited object types. The inheritance relation is represented by the dashed arrows outgoing from a base type to the subtype (note that these edges do not belong to the query graph, but the dotted ones are defined on their basis).

In the figure there are shown all the consecutive stages of the reduction that are explained in the following example instantions in this section.

Example 2. *In the graph of Example 2, the service* $CarRepair$ *is defined, where* $\{(c, Car)\} \subseteq inout_{CarRepair}$*. The edges* $v_{CarRepair} \rightarrow v_{Car}$ *and* $v_{Car} \rightarrow v_{CarRepair}$ *are added to the ontology graph. Moreover, the edges* $v_{CarRepair} \rightarrow v_{Truck}$ *and* $v_{Truck} \rightarrow v_{CarRepair}$ *are also added for* $Truck$ *being a subtype of* Car*. Note that our type system does not enable any restrictions for covariance of types, thus it is impossible at the type level to define a service which does not accept subtypes of some object type. This effect can be obtained in different ways (for example, by setting attributes), but when reasoning at the level of types, we need to consider all the subtypes for every object type.*

3.3.2 Query Graph

The ontology graph G_{Ont} (stored in the graph database) represents the ontology Ont. The next step is to extend G_{Ont} with a representation of a user query q by constructing a *query graph*, which contains also an *initial vertex* v_I and a *final vertex* v_F. The initial vertex is connected to the vertices

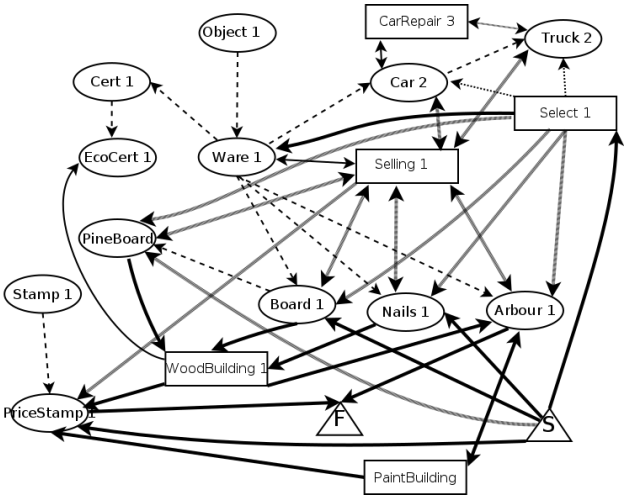

Figure 2: The query graph for the example. The oval and rectangle vertices correspond to object types and service types, respectively. The triangle vertices S and F model the start and final vertices, resp. Dashed edges represent the inheritance of object types (do not belong to any of the graphs used by our approach). The solid edges model consuming and producing objects by services. The dotted edges extend processing objects to derived object types. The double-sided arrows between a pair of vertices represent two single-sided arrows, in both directions, with the same properties (for example, bold and dotted). The number i means that the corresponding type belongs to Ont_i (and $Ont_{i+1}, Ont_{i+2}, \dots$)

corresponding to the types of the objects from the initial worldset. The vertices corresponding to the types and subtypes of the objects from the expected world of the user query are connected to the final vertex. v_I is also connected to the vertices corresponding to each service type having the list *in* and *inout* empty.

Definition 10 (Query graph)**.** *Given the ontology graph* $G_{Ont} = (V_{Ont}, E_{Ont})$ *and a user query specification* $q = (in_q, inout_q, out_q, pre_q, post_q)$*, where* $inout_q \cup out_q \neq \emptyset$[2]*. By the* query graph *we mean the graph* $G_q = (V_q, E_q)$*, where:*

- $V_q = V_{Ont} \cup \{v_I, v_F\}$,
- $E_q = E_{Ont} \cup \{(v_{\boldsymbol{t}}, v_F) \mid \boldsymbol{t} \in \mathcal{T}(out_q \cup inout_q)\} \cup$
  $\{(v_I, v_{\boldsymbol{t}}) \mid v_{\boldsymbol{t}} \in V_{\mathbb{T}} \wedge \ (\exists o \in (in_q \cup inout_q)) : \boldsymbol{t} \in type(o)\} \cup$
  $\{(v_I, v_s) \mid v_s \in V_{\mathbb{S}} \wedge \ in_s \cup inout_s = \emptyset\}$.

Notice that the subtypes are added only for the object types of the expected worldset which corresponds to the fact the the user accepts 'more' than he

[2]For the final world without any objects, the reduction would be not well-defined and result in no pruning at all. As will be explained later, the abstraction is guided by the object types from the final world.

requires. Clearly, one cannot assume that the user possesses 'more' (so no subtypes) than specified by the initial worldset.

Example 3. *In Fig. 2, the triangle shapes denoted with S and F model the initial node and final node of the query graph, respectively. The adjacent edges are defined according to Def. 10. Recall that the edges from the initial vertex are outgoing to the vertices for object types from the initial world of the user query, and to the service Select, which produces objects out of nothing. The bold edges (solid and dotted) belong to the paths between the initial and final vertices, restricted to the depth* 3 *(so their length is bound by* $2 \cdot 3 + 2 = 8$*. The graph database returns this set of paths as a result of the database query.*

In Example 1, the edge from v_I *to the node* $v_{PineBoard}$ *is added to build the query graph, because PineBoard is a subtype of the object type Board.*

3.3.3 Query k-subgraph: pruning Query Graph

Our next step consists in pruning the query graph leaving only its subgraph (called the *query k-subgraph*) induced by all the paths of length k from the initial vertex to the final vertex. This subgraph is produced as the result of a database query to the graph database storing the query graph. In this query the depth is given by $2 \times k + 2$ to reflect the fact that a plan of length k corresponds to a path of length $2 \times k + 2$ in the query graph because of its construction.

Below we formalize the above concept.

Definition 11 (Query k-subgraph)**.** *Let* $G_q = (V_q, E_q)$ *be the query graph and* $k \in \mathbb{N}$*. The* query k-subgraph $G^k_{qs} \subseteq G_q$ *is the result of executing the database query* $Q^{2k+2} = (v_I, v_F, 2k + 2)$ *onto the query graph, where* v_I, v_F *are the initial and final vertices of* G_q*, respectively.*

Example 4. *In Fig. 2, bold lines correspond to the edges belonging to the query* 3*-subgraph. The node set consists of all the adjacent vertices.*

In order to define formally the ontology pruned we need the notion of *supertypes* of a set object types. Formally, for $T \subseteq \mathbb{T}$ we define $superTypes(T) = \bigcup_{\boldsymbol{t} \in T} \{\boldsymbol{t'} \in \mathbb{T} \mid (\boldsymbol{t'}, \boldsymbol{t}) \in Ext^*\}$.

The pruned ontology is the final result of the graph reduction. As a special case, empty sets of service and object types are returned if there exists an object type from the expected world which cannot be produced (there is no path in the query subgraph leading to the node modeling it, and the same is true for all its subtypes)[3]

[3]Note that according to the semantics of abstract planning, all the object types from the expected world need to be present in the expected world. If the user requests an alternative of object from this set, it can be done by providing several queries or by specifying a postcondition.

We say that Ont_k is an *empty ontology* if it has empty sets of services and object types. Sometimes, empty ontologies can be identified by simple property of the query k-subgraph:

Definition 12 (query k-subgraph generating empty ontology)**.** *Let $G_{qs} = (V_{qs}, E_{qs})$ be the query k-subgraph. We say that G_{qs} generates an empty ontology if $\exists o \in \mathbb{O}$ such that $o \in inout_q \cup out_q$ and for $\forall \boldsymbol{t} \in \mathcal{T}(\{o\})$, there are no incoming transitions to $v_{\boldsymbol{t}} \in V_{qs}$.*

The pruned ontology is formally defined as follows:

Definition 13 (k-Reduced ontology)**.** *Let $Ont = (\mathbb{S}, \mathbb{T}, Ext)$ be an ontology and $G_{qs} = (V_{qs}, E_{qs})$ be the query k-subgraph. By the* k-reduced ontology *we mean the ontology $Ont_k = (\mathbb{S}_k, \mathbb{T}_k, Ext_k)$, which is empty iff G_{qs} generates an empty ontology, and otherwise we have:*

- $\mathbb{S}_k = \{s \in \mathbb{S} \mid v_s \in V_{qs}\}$,
- $\mathbb{T}_k = superTypes(\bigcup_{s \in \mathbb{S}_k} \mathcal{T}(in_s \cup inout_s \cup out_s))$,
- $Ext_k = Ext \cap (\mathbb{T}_k \times \mathbb{T}_k)$.

Notice that the k-reduced ontology contains all the service types and all the object types corresponding to the vertices of G^k_{qs}. In addition, it contains all the supertypes of the subtypes of the sets of the input and output object types of each its service type.

Example 5. *In Fig. 2, for every node there is shown a number i, meaning that the type represented by this node belongs to the reduced ontology Ont_i (and $Ont_{i+1}, Ont_{i+2}, \dots$). Note that some service and object types (for example, the node* Car*, its descendant* Truck*, and service* ServiceCar *operating on them) are added for bigger depths, when a plan is found for smaller depths.*

Service type PaintBuilding is included to the reduced ontology because of a self-loop for the Arbour type.

In Definition 13, supertypes of object types are added with different motivation than subtypes in Definitions 9 and 10. In contrast to subtypes, a supertype cannot substitute an object type in abstract planning. However, at this stage all the types relevant for planning have been identified. We want the reduced ontology to be complete and well-defined, i.e. every object type should have all its predecessors (i.e., supertypes) up to *Object* type. This is important both from the perspective of theoretical correctness and in order to ensure that planers could work correctly on the reduced ontology.

Example 6. *In Example 1, G^k_{qs} (with $k = 2$) contains (among others) the vertices v_{Board}, v_{Nails}, and v_{Arbour}. The vertices v_{Ware} and v_{Object} do not*

occur in G^k_{qs}, but these types are added to the reduced ontology being supertypes of some of the vertices listed above. The vertex $v_{EcoCert}$ does not belong to G^k_{qs}, but the type EcoCert and its predecessor Cert are elements of the 2-reduced ontology, because EcoCert is the type of the service BuildArbour out set.

We assume that the out_q list of the user query q is not empty as otherwise the reduced ontology would be equal to the original one. This follows from the fact that then one would need to consider how every object type can be produced, resulting in no reduction at all. On the other hand, if the input lists in_q and $inout_q$ of the user query are empty, then the object types of out_q can still be potentially produced by service types which create objects "out of nothing", provided such service types are in the ontology.

3.4 Correctness of the reduction

Now we prove that the reduced ontology preserves all the user query solutions. To this aim we define *object type derivation sequences (OTDSs)*, showing how the object types of the expected worldset are produced in each user query solution by a sequence of service types interleaved with object types. For every user query solution, we define the set of all the OTDSs producing all the object types of the expected world. We show that each service type s of the user query together with the set of all the types and subtypes of $in_s \cup inout_s \cup out_s$ is present in some OTDS. Finally, we show that each OTDS can be mapped to the path in the query k-graph, thus proving that all the object and service types needed to preserve all the user query solutions are present in the reduced ontology.

Definition 14 (Object type derivation sequence (OTDS))**.** *Given an ontology Ont, a user query q, a solution of q* $\text{seq} = \big((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k})\big)$ *of length k, and $m \leq k$. By an* object type derivation sequence (OTDS for short) *we mean a sequence $otds = (T^0, s^1, T^1, s^2, T^2 \ldots, T^{m-1}, s^m, T^m)$, satisfying the following conditions:*

- $s^1, \ldots, s^m \in \{s_1, \ldots, s_k\}$.
- $T^0, \ldots, T^m \subseteq \mathbb{T}$, *where* $|T^i| = 1$ *for* $1 \leq i \leq m$, $|T^0| = 1$ *if* $in_{s^1} \cup inout_{s^1} \neq \emptyset$, *and* $T^0 = \emptyset$ *otherwise*[4],
- $T^m \subseteq \mathcal{T}(inout_q \cup out_q)$,
- $T^0 \subseteq \mathcal{T}(in_q \cup inout_q)$,

[4]Inclusion of empty sets can occur in the following definitions. For the sake of brevity, we do not distinguish this special case.

- *there is a monotonic function* $tr : \{1, \ldots, m\} \to \{1, \ldots, k\}$ *that maps the index of every service type of otds into the index of an element of seq, such that for each* $1 \leq i \leq m$ *and* $1 \leq j \leq k$ *if* $tr(i) = j$*, then the following conditions hold:*
  - $s^i = s_j$,
  - $T^i \subseteq \mathcal{T}(out_{s_i} \cup inout_{s_i})$,
  - $T^{i-1} \subseteq \mathcal{T}(in_{s_i} \cup inout_{s_i})$.

By $OTDS_O(Ont, q, k)$ we denote all the object type derivation sequences the given ontology, user query and any plan of the length equal or less than k, producing the object types from the set $\mathcal{T}(O)$, for $O \in \mathbb{O}$.

Note that the definition of OTDS is based on an abstract plan, but the mapping refers only to the sequence *seq*, without any reference to the valuations. This can be explained by the fact that our reduction works at the level of types, so OTDS preserves the mapping of object types to service types in a sequence, assuming that there are valuations making this sequence a valid plan.

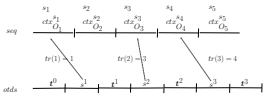

Figure 3: Mapping of *odts* to *seq*.

Example 7. *In Fig. 3 we show a context abstract plan with a corresponding sequence* $seq = \big((s_1, ctx^{s_1}_{O_1}), \ldots, (s_5, ctx^{s_5}_{O_5})\big)$*, and* $otds \in OTDS_{O_5}(Ont, q, 5)$ *mapped to it.* $O_5 = \{n : Nails, b : Boards, a : Arbour, e : EcoCert, p_1, p_2, p_3, p_4, p_5 : PriceStamp\}$*. Concerning the remaining object sets we only state, for the sake of brevity, that* $O_1, \ldots, O_4 \subseteq O_5$*.*

- $s_1 = Transport$, $\{n : Nails, p_1 : PriceStamp\} \subseteq O_1$,
  $ctx^{s_1}_{O_1}(inout_{s_1}) = \{n : Nails\}$,
  $ctx^{s_1}_{O_1}(out_{s_1}) = \{p_1 : PriceStamp\}$,
- $s_2 = Transport$, $\{b : Boards, pb : PriceStamp\} \subseteq O_2$,
  $ctx^{s_2}_{O_2}(inout_{s_2}) = \{b : Boards\}$,
  $ctx^{s_2}_{O_2}(out_{s_2}) = p_2 : PriceStamp$,
- $s_3 = WoodBuilding$, $\{e : EcoCert, n : Nails, b : Boards, a : Arbour, p_3 : PriceStamp\} \subseteq O_3$,
  $ctx^{s_3}_{O_3}(in_{s_3}) = \{e : EcoCert\}$,
  $ctx^{s_3}_{O_3}(inout_{s_3}) = \{b : Boards, n : Nails\}$,
  $ctx^{s_3}_{O_3}(out_{s_3}) = \{a : Arbour, p_3 : PriceStamp\}$,

- $s_4 = Selling$, $\{a : Arbour, p_4 : PriceStamp\} \subseteq O_4$,
  $ctx^{s_4}_{O_4}(inout_{s_4}) = \{a : Arbour\}$,
  $ctx^{s_4}_{O_4}(out_{s_4}) = \{p_4 : PriceStamp\}$,
- $s_5 = Transport$, $\{b : Boards, p_5 : PriceStamp\} \subseteq O_5$,
  $ctx^{s_5}_{O_5}(inout_{s_5}) = \{b : Board\}$,
  $ctx^{s_5}_{O_5}(out_{s_5}) = \{p_5 : PriceStamp\}$.

The mapping tr is defined as follows:

- $tr(1) = 1$
- $tr(2) = 3$
- $tr(3) = 4$

It determines the following OTDS: $T^0 = \{Nails\}$, $s^1 = Transport$, $T^1 = \{Nails\}$, $s^2 = WoodBuilding$, $T^2 = \{Arbour\}$, $s^3 = Selling$, $T^3 = \{Arbour\}$.

Pruning may reduce some plans to the minimal form, where every service is relevant for the composition. This notion is captured by the following definition:

Definition 15. *Minimal solution Let* seq $= \big((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k})\big)$ *be a user query solution for q. We say that* seq *is a* minimal solution*, if no strict subsequence of* seq *is a user query solution.*

The following lemma states that all the service and object types occurring in all the object type derivation sequences for every valid plan, are represented in the k-query subgraph.

Lemma 1. *Given an ontology Ont and a user query q, a minimal solution of q* seq $= \big((s_1, ctx^{s_1}_{O_1}), \ldots, (s_k, ctx^{s_k}_{O_k})\big)$ *of length k. The k−reduced ontology* Ont_k *contains every service type* s_i *and every object type* $\boldsymbol{t} \in \mathcal{T}(O_k)$.

Proof. The proof is by mathematical induction. The induction will be over the length *seq*, starting from its end, that is we start counting from the final state, towards the initial one. We do not apply the induction directly to the minimal solutions, which cannot be made shorter. Instead, the reduction will operate on reversed prefixes of minimal sequences. Let $seq_{\downarrow l} = \big((s_{k-l+1}, ctx^{s_{k-l+1}}_{O_{k-l+1}}), \ldots, (s_k, ctx^{s_k}_{O_k})\big)$ be the reversed prefix of *seq* for $1 \leq l \leq k$ (for $l = k$ we have the whole sequence).

Basis: $l = 1$ We have $seq_{\downarrow 1} = \big((s_k, ctx^{s_k}_{O_k})\big)$. For each assignment of objects from O_k to the input and output arguments of s_k, the corresponding OTDSs have the following form: $otds = (T^{k-1}, s^k, \{\boldsymbol{t}^k\})$, for $s_k = s^k$, $\boldsymbol{t}^k \in$

$\mathcal{T}(inout_{s_k} \cup out_{s_k})$ and $T^{k-1} \subseteq \mathcal{T}(in_{s_k} \cup inout_{s_k})$. It is directly represented by the following paths in $G_{qs}(Ont, q, k)$: $v_I \to v_{s^k} \to v_{\boldsymbol{t}^k} \to v_F$ if $T^{k-1} = \emptyset$, or $v_I \to v_{\boldsymbol{t}^{k-1}} \to v_{s^k} \to v_{\boldsymbol{t}^k} \to v_F$ otherwise, for $T^{k-1} = \{\boldsymbol{t}^{k-1}\}$.

For some objects produced by s^k but not belonging to $inout_q \cup out_q$, their object types need not to be in any OTDS. However, all the object types produced by every service in the reduced ontology are also explicitely added to it (see Def. 13).

Inductive case: Assume that the statement holds for $G_{qs}(Ont, q, l)$. We have that s_i and all the object types from $\mathcal{T}(O_i)$ are represented in $G_{qs}(Ont, q, l)$, for $1 \leq i \leq l$. We need to show that s_{k-l} and all the object types from $\mathcal{T}(O_{k-l})$ also are in $G_{qs}(Ont, q, l+1)$.

s_{k-l} belongs to at least one OTDS, because the plan is minimal, thus every service processes some object from the final world. Otherwise, it could be removed from the plan, which would not be minimal. Objects are not discarded, thus every object produced by any service in the plan is in the final world. For a service, the only way to produce or change objects is to have them in its output lists, as the services have no side effects (such as modifying global variables, changing variables accessed by read-only pointers, etc).[5]

Every object type occurring in the plan is either introduced in the initial world of the user query, or produced by a service. Later it can be possibly processed by other services, or enable producing other objects (if belongs to *inout* or *in* list, respectively). This sequence of transformations is captured by a corresponding OTDS.

The length of ODTSs mapped to $seq_{\downarrow k+1}$ is bound by $k+1$, as there are $k+1$ services in this prefix and every service in every OTDS is mapped to at most one of these services. Let assume that s^{k-l} belongs to $otds = (T^0, s^1, \{\boldsymbol{t}^1\} \ldots, \{\boldsymbol{t}^{m-1}\}, s^m, \{\boldsymbol{t}^m\})$. There is the corresponding path $v_I \to v_{\boldsymbol{t}^0} \to v_{s^1} \to v_{\boldsymbol{t}^1} \to \cdots \to v_{s^m} \to v_{\boldsymbol{t}^m} \to v_F \in G_{qs}$ if $inout_{s^1} \cup in_{s^1} \neq \emptyset$, or $v_I \to v_{s^1} \to v_{\boldsymbol{t}^1} \to \cdots \to v_{s^m} \to v_{\boldsymbol{t}^m} \to v_F \in G_{qs}$ otherwise. From the definition of the reduced ontology it follows that it contains both s^{k-l} and all the object types processed by it and belonging to $inout_{s^{k-l+1}} \cup in_{s^{k-l+1}}$.

Concerning the objects produced by s^{k-l} but not belonging to $inout_{s^{k-l+1}} \cup in_{s^{k-l+1}}$, their types need not to be in any OTDS, and in consequence, in any path of G_{qs}. However, all the object types produced by every service added to the reduced ontology are also explicitely included (see Def. 13), with their subtypes.

Concerning the inheritance, note that we need only to ensure that every OTDS is represented in the query subgraph, and paths in the query subgraph directly correspond to OTDSs, with respect to the subtypes of every object type. This is guaranteed by Def. 9, by adding the edges between nodes

[5]This is why there are no services without output in *seq*, as they would be removed. We have no casting nor dynamic transformations of object nor service types, thus object types do not change between processing by services.

representing services and nodes representing subtypes of every object type processed by these services.

□

The corollary from the above lemma states that the pruned ontology can be used as an input to an abstract planner, and for every user query q and the given depth k, a solution for q of the length k will exist in the k-reduced ontology Ont_k if and only if it exists in the complete ontology[6] Ont.

Corollary 1. *For every service type s_i and every object $o \in O_j$ occuring in* seq $= \big((s_1, ctx_{O_1}^{s_1}), \ldots, (s_k, ctx_{O_k}^{s_k})\big)$ *for $1 \leq i \leq k$, and $o \in O_j$ for some $1 \leq j \leq k$, we have that these services and objects are represented in the reduced ontology, i.e. $s_i \in \mathbb{S}$, $superTypes(\{type(o)\}) \subseteq \mathbb{T}$ and Ext is the inheritance relation from the full ontology restricted to the types present in the reduced one, for $Ont_k(Ont, q, k) = (\mathbb{S}, \mathbb{T}, Ext)$.*

Proof. By Lemma 1, every service and object type from each minimal transformation sequence are present in Ont_k. By its construction, according to Def. 13, all the supertypes are added for every of these types, so the type system in the reduced ontology is complete. □

Note that the types from the user query, with their supertypes, need not to be added explicitely. The explanation is that either they are added because there exists at least one path in $G_{qs}(Ont, q, l)$ to the correponding node (or its subtype), or there is no such a path and we return an empty ontology.

While the reduced ontology is formally sound and complete, we know in advance that some of the object and service types added in Def. 13 will not contribute to any valid plan. For service types, this can be true because of some object types processed by them, as explained below. Every object type represented by Ont_R falls into one or more of the following categories:

1. is an input or output for a service type, and belongs to some path in the query k-subgraph.
2. is an output for a service type. It does not belong to any path in the query k-subgraph, but has been added explicitely by Def. 13,
3. is an input for a service type. It does not belong to any path in the query k-subgraph, but has been added explicitely by Def. 13.

The object types which satisfy only condition 3. do not contribute to any valid plan (because the service type cannot execute as it lacks some input object types, which have been added to ontology but there is no way

[6] As will be shown in the example, the solution is guaranteed to exist in Ont_k, but sometimes it may be found in Ont_i for $i < k$.

to provide them) and can be removed by postprocessing, along with the corresponding services types.

From a practical perspective, reducing the length of plans is an advantage. As it will be shown in the section describing experimental results, the planners are very sensitive to the length of plans. In particular, the performance of the SMT-based planners degrades very quickly when the length of plans grows. Preserving minimal plans decreases also the number of services in the reduced ontology.

Example 8. *Assume that in addition to those defined in Example 7 we have the following services for some* $O_6, O_7, O_8 \subseteq \mathbb{O}$*:*

- $s_6 = Select$, $\{a : Arbour, p_6 : PriceStamp\} \subseteq O_6$,
  $ctx^{s_6}_{O_6}(inout_{s_6}) = \{a : Arbour\}$,
  $ctx^{s_6}_{O_6}(out_{s_6}) = \{p_6 : PriceStamp\}$,
- $s_7 = Select$, $\{t : Truck, p_7 : PriceStamp\} \subseteq O_7$,
  $ctx^{s_7}_{O_7}(inout_{s_7}) = \{t : Truck\}$,
  $ctx^{s_7}_{O_7}(out_{s_7}) = \{p_7 : PriceStamp\}$,
- $s_8 = Selling$, $\{a : Arbour, p_8 : PriceStamp\} \subseteq O_8$,
  $ctx^{s_8}_{O_8}(inout_{s_8}) = \{a : Arbour\}$,
  $ctx^{s_8}_{O_8}(out_{s_8}) = \{p_8 : PriceStamp\}$,

Now we consider some examples of transformation sequences and whether are they preserved in the reduced ontologies:

1. $\mathrm{seq}_1 = \big((s_6, ctx^{s_6}_{O_6}), (s_7, ctx^{s_7}_{O_7}), (s_8, ctx^{s_8}_{O_8})\big)$
   seq_1 *will not be preserved in the reduced ontology* Ont_1*. The reason is that there is no path in the query* 1*-subgraph going through the* $Truck$ *object node (see Fig. 2). Instead, the transformation sequence* $\mathrm{seq}_{1'} = \big((s_6, ctx^{s_6}_{O_6}), (s_8, ctx^{s_8}_{O_8})\big)$ *will be preserved in* Ont_1*, satisfying the user query.*

   However, seq_1 *is preserved in* Ont_2*.*
2. $\mathrm{seq}_2 = \big((s_3, ctx^{s_3}_{O_3})\big)$*.* seq_2 *is preserved in* Ont_1*.*
3. $\mathrm{seq}_3 = \big((s_3, ctx^{s_3}_{O_3}), (s_4, ctx^{s_4}_{O_4})\big)$*.* seq_3 *is preserved in* Ont_1*. This case is interesting, because the solution is determined by two sequences in the query* 1*-subgraph, one of which does not correspond to any valid object type transformation. In particular, we have (among others) the following paths:* $v_I \to v_{Nails} \to v_{Selling} \to v_{Arbour} \to v_F$*, and* $v_I \to v_{Select} \to v_{Arbour} \to v_F$*. The first one does not represent any*

transformation of object types from a valid plan (contrary to the path $v_I \rightarrow v_{Select} \rightarrow v_{Arbour} \rightarrow v_{Selling} \rightarrow v_{Arbour} \rightarrow v_F$*, which occurs for* $k \geq 2$*), but it includes the Selling service type to the reduced ontology. In the future, we plan to make the database search more precise, so that it would follow exact object transformations, without taking all the object types (and their descendants) belonging to inout list of a service type.*

It can be easily seen that the presented abstraction is in fact an over-approximation. Some of the paths in the query k-subgraph returned by the database can have no corresponding object type derivation sequences for any valid plan.

Example 9. *In Example 1, object type EcoCert along with its ancestor Cert are added to the reduced ontology despite the fact that they do not participate in any potentially valid plan for the specified depth* $k = 2$*. The explanation is that we need to assure that the service type WoodBuilding has all the required input and output types in the reduced ontology (so EcoCert is added) and that every object type needs to have all its predecessors in the ontology (so Cert is added).*

When, for every expected world, there is no path to at least one of its object types in the query subgraph, this means that no valid abstract plan exists. However, the existence of such paths for all the object types of an expected world does not guarantee that there is a valid abstract plan. This is because the graph approach works at the level of types, and does not take into account the issues such as checking pre- and postconditions, and providing enough objects for cardinality constraints. Checking the pruned ontology by an abstract planner is still needed, but usually the scope of this search will be significantly reduced. Thus, the method is complete and sound, and every valid plan will be found in the query subgraph for the chosen depth.

Note also that the self-loops in Definition 9 of the ontology graph are necessary despite the fact that possible setting of object attributes by the corresponding services is irrelevant at the level of matching types. What matters is that the resulting OTDSs can decide about including these services to the pruned ontology.

The restriction on depth is introduced to prevent queries from searching the whole ontology and to keep the search local. All PlanICS abstract planners perform only the bounded-depth searches. The intuition is that we expect the user queries to be local in the sense of involving only a restricted number of services. In general, there is no bound on the lengths of plans, as some combination of services can set and unset selected attributes of objects, resulting in valid plans of infinite lengths.

3.4.1 Implementing the algorithm in the graph database

As we have described above, the role of the graph database is to represent the ontology graph and, for every user query, to extend it to the query graph and find the query subgraph. We used the graph database Neo4j, working in the standalone mode (that is, without the server installation, but run over a Java API communicating with the database, performing operations such as adding vertices and edges, and labeling them). The graphs are directly represented by the graph database in accordance with its semantics model, so no transformations of any kind are needed.

The general way of interaction with the database is as follows: first the ontology graph is stored in the database. It can be expected to be of a significant size, but its construction is performed only once, independently of user queries. Then, for each user query, the query graph is constructed by adding the initial and final vertex, with respective edges. Next, the database query is entered to Neo4j, expressed in the language *Cypher*. It is independent of the user query and its meaning is: *find all the paths of the depth at most k, between the start and the final vertices.* Finally, the start and final vertices with their edges are removed from the database, and the system is ready for the next user query.

Example 10. *The search for query k-subgraph is expressed by the following* Cypher *query, for* $k = 8$. *node*(1) *and node*(2) *correspond to the initial node* v_I *and final node* v_F*, respectively:*

```
start a=node(1), b=node(2)
match p=a-[r:PROCESSED*0..8]->b
with p return p;
```

PROCESSED is a label of all the edges. It is introduced to distinguish edges with different meanining, for example representing inheritance of types.

The database returns the query subgraph.

3.4.2 Postprocessing the query subgraph (optional)

The set of paths in a query subgraph returned by the database may contain some vertices corresponding to services which cannot execute in the pruned ontology, because some of the objects from their input lists are missing from the subgraph. Postprocessing recursively removes all the occurences of such services. It also removes the objects not associated with any of the remaining services. We can also remove vertices representing the services, which have only their *inout* lists nonempty.

3.4.3 Generating the pruned ontology from query subgraph, and testing the query

Finally, the pruned ontology is generated so that it contains all the service and object types corresponding to the vertices of the query subgraph. It is saved in a file. The user query is tested on it using an abstract planner for the depth k being the parameter of the *Cypher* query. Note that any *Planics* abstract planner can be applied, including all those presented in this paper, what will be shown in the section presenting the experimental results.

4 Example explained in detail

The benchmarks tested in this paper are generated automatically for predefined parameters, using the same generator as in [12] and other papers about PlanICS. Names of services and objects are random strings. Several papers about abstract planning in PlanICS describe more human-readable examples from more realistic domains. In this Section we describe in detail the benchmark number 3. There are 256 instances of service types in the ontology. Only one abstract plan exists for the following user query:

```
in=Dxvcr dxvcr1
inout=Opufo opufo1
out=Vpumc vpumc1, Hemjg
    hemjg2, Wldue wldue3
pre=isSet(dxvcr1.xii) and
    isSet(opufo1.syk) and
    isSet(dxvcr1.hvf) and
    isSet(opufo1.gty)
post=isSet(hemjg2.efx) and
    isSet(opufo1.syk) and
    isNull(opufo1.gty) and
    isNull(wldue3.zuh) and
    isNull(vpumc1.ihy) and
    isNull(wldue3.txm)
```

That is, there are two objects in the initial world, one of which is expected to be also in the expected world. Additional there objects are to be produced. There are some requirements on attributes of those objects.

In Fig. 4 it is shown the query subgraph, as displayed by the Neo4j visualization module. Silver and purple nodes represent the services and the objects, respectively. The red nodes labeled with 1 and 2 stand for the initial and final node. For this example, this graph directly corresponds to the (only) abstract plan in the obvious way: every path from the initial to the final node corresponds to a partial order of the plan.

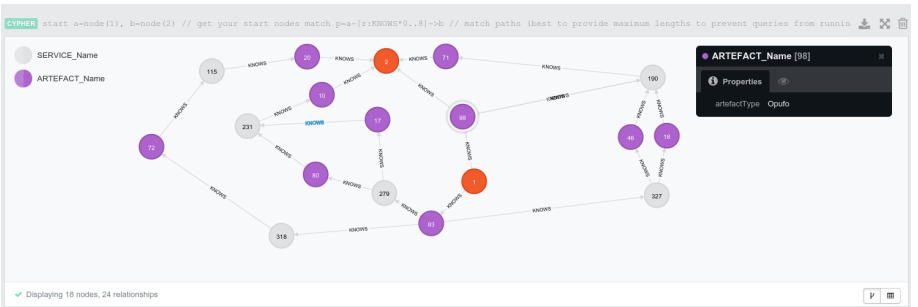

Figure 4: The multi-plan graph for Ontology 3. Explanations are provided in the text.

5 Experimental results

For the experiments we used the graph database Neo4j, version 2.1.6, running on Linux. We do not report on the technical issues like setting up indices, optimizing storage, etc. In fact, the default configuration has been applied. The experimental results are shown in Table 1. The experiments are performed in an analogous way as in [15] and other papers describing the PlanICS abstract planners, i.e. we use automatically generated benchmarks and user queries described there. Ontologies are pruned on the basis of respective query subgraphs by manipulating the SMT planner structures in memory, without generating the files for pruned ontologies.

It can be seen that pruning the search space gives very significant reductions in verification time. It is also worth noticing that the examples have the following structure: every three consecutive benchmarks generate the same plan (or plans), but there are different numbers of services not participating in these plans. Planning times grow respectively for the SMT planner without pruning, and after filtering by the graph database remain similar, of the same order. Differences can be traced back to different ordering of services when pruning the ontologies to match the subgraphs returned by the graph database.

Note that in a realistic scenario of the real-world system application, the advantage of our approach over other planners could be even more significant, because the full ontology could be encoded only once in the graph database. Then, for every user query only the initial and final nodes would need to be added, and the respective search be performed. The existing planners need to encode the whole ontology for every search.

6 Conclusions and Future Work

In the paper we proposed an original approach significantly improving the effectiveness of abstract planning phase in the PlanICS web composition sys-

Table 1: Experimental results for GraphDB. Execution times are given in seconds. **First** refers to time required to find the first solution. **Next** is time after which all the remaining solutions have been found (if exist). **UNSAT** is time in which the solver determines that no more plans exist. *to* means timeout, set to 1000 seconds.

example	SMT			GraphDB+SMT			
	First	**Next**	**UNSAT**	k	**First**	**Next**	**UNSAT**
1	2.74	-	4.11		0.17	-	0.25
2	14.46	-	18.21	8	0.2	-	0.44
3	15.01	-	17.43		0.2	-	0.52
4	4.25	12	8		0.68	1.8	3
5	6.82	29.9	25	8	0.75	2.15	2.23
6	15.32	32.41	53.39		0.73	2.02	2.93
7	5.98	-	27		0.57	-	4.1
8	33.18	-	81	11	0.45	-	5.05
9	101	-	271		0.46	-	5.46
10	15	59	95		5.48	22.17	69
11	97	156	392	14	12.46	31	99
12	to	to	to		7.78	16	128
13	75.22	-	225		6.68	-	20
14	140.7	-	812.3	11	3.88	-	70.34
15	to	-	to		5.8	-	57.16
16	to	-	to		29.9	-	846.4
17	to	-	to	14	24.1	-	897.2
18	to	-	to		29.3	-	902.1

tem. It consists in modeling the planing as a graph and applying a graph database for pruning the search performed by an abstract planner. The experiments performed on planning benchmarks confirmed that pruning makes planning times shorter, sometimes by several orders of magnitude. The overhead caused by database search is very small. An additional advantage is enabling the way of visualizing structures of ontologies, what can be helpful in several applications, such as doing optimizations, analyzing which services and objects are used most often, etc.

As a next step we plan to implement the whole abstract planning process by means of graph algorithms. This is motivated by the fact that in the cases where the pruned ontologies are still to hard for existing planners, there is the space for improvements and it is likely that graph algorithms can profit from the ability to exploit the structure of ontologies and queries. We also plan to extend the benchmarks, resulting in more complex abstract plans, with several branchings between partial orders, and with inheritance of objects and services types.

References

[1] *A Graph-Based Web Service Composition Technique Using Ontological Information* (2007).

[2] Ambroszkiewicz, S. Entish: A language for describing data processing in open distributed systems. *Fundam. Inform. 60*, 1-4 (2004), 41–66.

[3] Angles, R., and Gutierrez, C. Survey of graph database models. *ACM Comput. Surv. 40*, 1 (Feb. 2008), 1:1–1:39.

[4] Deng, S., Wu, B., Yin, J., and Wu, Z. Efficient planning for top-k web service composition. *Knowledge and Information Systems 36*, 3 (2013), 579–605.

[5] Doliwa, D., Horzelski, W., Jarocki, M., Niewiadomski, A., Penczek, W., Pólrola, A., Szreter, M., and Zbrzezny, A. Planics - a web service composition toolset. *Fundam. Inform. 112*, 1 (2011), 47–71.

[6] Elmaghraoui, H., Zaoui, I., Chiadmi, D., and Benhlima, L. Graph based e-government web service composition. *CoRR abs/1111.6401* (2011).

[7] Hashemian, S. V., and Mavaddat, F. A graph-based framework for composition of stateless web services. In *ECOWS* (2006), IEEE Computer Society, pp. 75–86.

[8] Knapik, M., Niewiadomski, A., and Penczek, W. Generating none-plans in order to find plans. In *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings* (2015), pp. 310–324.

[9] Li, X., Zhao, Q., and Dai, Y. A semantic web service composition method based on an enhanced planning graph. ICEE, 2288-2291(2010), 2010.

[10] Mahmoud, C. B., Bettahar, F., Abderrahim, H., and Saidi, H. Towards a graph-based approach for web services composition. *CoRR abs/1306.4280* (2013).

[11] Neo4j. *Neo4j - The World's Leading Graph Database*. 2012.

[12] Niewiadomski, A., and Penczek, W. Smt-based abstract temporal planning. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, co-located with 35th International Conference on Application and Theory of Petri Nets and Concurrency (PetriNets 2014) and 14th International Conference on Application of Concurrency to System Design (ACSD 2014), Tunis, Tunisia, June 23-24, 2014.* (2014), pp. 55–74.

[13] Niewiadomski, A., Penczek, W., and Półrola, A. Abstract Planning in PlanICS Ontology. An SMT-based Approach. Tech. Rep. 1027, ICS PAS, 2012.

[14] Niewiadomski, A., Penczek, W., and Skaruz, J. SMT vs genetic algorithms: Concrete planning in PlanICS framework. In *Proceedings of CS&P, Warsaw, Poland* (2013).

[15] Niewiadomski, A., Penczek, W., and Skaruz, J. A hybrid approach to web service composition problem in the planics framework. In *Mobile Web Information Systems - 11th International Conference, MobiWIS 2014, Barcelona, Spain, August 27-29, 2014. Proceedings* (2014), pp. 17–28.

[16] Niewiadomski, A., Skaruz, J., Penczek, W., Szreter, M., and Jarocki, M. SMT versus genetic and openopt algorithms: Concrete planning in the planics framework. *Fundam. Inform. 135*, 4 (2014), 451–466.

[17] Pokorny, J. Nosql databases: A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services* (New York, NY, USA, 2011), iiWAS '11, ACM, pp. 278–283.

[18] Robinson, I., Webber, J., and Eifrem, E. *Graph Databases*. O'Reilly Media, Inc., 2013.

[19] Shetty, S., R, S. P., and Sinha, A. K. Article: A novel web service composition and web service discovery based on map reduce algorithm. *IJCA Proceedings on International Conference on Information and Communication Technologies ICICT*, 4 (October 2014), 41–45. Full text available.

[20] Talantikite, H. N., Aissani, D., and Boudjlida, N. Semantic annotations for web services discovery and composition. *Computer Standards & Interfaces 31*, 6 (2009), 1108 – 1117.

[21] Web Service Modelling Ontology D2v1.0. http://www.wsmo.org/2004/d2/v1.0/, 2004.