Regular Open APIs

Diego Calvanese Faculty of Computer Science, Free Univ. of Bozen-Bolzano, Italy
Giuseppe De Giacomo, Maurizio Lenzerini Dip. di Informatica e Sistemistica, Univ. di Roma "La Sapienza", Italy
Moshe Y. Vardi Department of Computer Science, Rice University, Houston, TX, U.S.A.

Abstract

Open APIs are software intermediaries that make it possible for application programs to interact with data and processes, which can both be viewed as forms of services. In many scenarios, when one wants to obtain or publish a new service, one would like to check whether the new functionality can simply be obtained by suitably composing existing services. In this paper we study this problem by distinguishing between the two forms of services, that we call data-centric and process-centric, respectively. In the former, each API is an abstraction of a query specified on a data source, and composition amounts to building a new query by using the available APIs as views over the data. In the latter, each API abstracts a process made up by sequences of atomic actions, and composition means realizing a new process by suitably using the APIs exposed by the available services. We make the assumption that the semantics of services is specified by means of one of the most basic formalisms used in Computer Science, namely, regular languages. As a result, we get a rich analysis framework, where compositions shows similarities to conformant and conditional planning. We describe composition principles and automated synthesis techniques for each of the two settings.

1 Introduction

An Open API (Application Programming Interface), or simply API, is a software intermediary that makes it possible for application programs to interact with each other and cooperate (Benslimane, Dustdar, and Sheth 2008). Specifically, an API specifies the mechanism for invoking a software service, i.e., an abstraction of a specific software functionality, exposed to clients, while hiding internal details of the application. Recent years have witnessed a significant interest in service modeling and service composition (Bouguettaya, Sheng, and Daniel 2014). Services are modeled by specifying the functionality they realize, and by associating with each of them an API that exposes the service to clients. With a set of services already available, new functionalities can thus be obtained by suitably composing the existing services. The composition is specified by using suitable operators, and its result is a new service, again abstracted by means of an API. Given a service to realize, called *target service*, how do we check whether it can be obtained by a suitable composition of the available services? If the answer to the above question is positive, how is the composition determined and constructed? And additionally, "can we build the composition automatically"? The last question characterizes what is called the *automatic service-composition problem* (Berardi et al. 2003; Hull 2005; Bartalos and Bieliková 2011; De Giacomo, Patrizi, and Sardiña 2013; Bouguettaya, Sheng, and Daniel 2014).

The above simple, general scheme is a good abstraction of many approaches to automatic service composition. Indeed, in this paper we illustrate the richness of the automatic composition problem, by considering several relevant settings, and describing principles and techniques for each of them. Since our goal in this work is to address fundamental issues of automatic service composition, we make the assumption that the semantics of services is specified by means of one of the most basic formalisms used in Computer Science, namely, *regular languages*. More precisely, we assume that associated with each service, be it one of the available services, or the target service to realize, we have a finite-state automaton, specifically a *deterministic finite-state automaton* (DFA), specifying its behaviour. Correspondingly, we call the APIs associated with services, *regular APIs*. Obviously, our analysis can be extended to take into account different, and more powerful mechanisms for modeling services.

In classifying the various settings, the main distinction we make is between the *data-centric view* and the *process-centric view* of services. Roughly speaking, in the data-centric view, each service corresponds to view, i.e., a query specified over a hidden database, and its invocation triggers the execution of such query, which returns the corresponding result. Service-composition means here building a new query by using the available services as *views* over the database (Navathe, Elmasri, and Larson 1986). In contrast, in the process-centric view, APIs represent processes made up by sequences of inaccessible atomic actions, and service composition means realizing a target process, again specified in terms of the inaccessible atomic actions, by exploiting the available APIs.

The goal of this paper is to present techniques and computational complexity analyses for the automatic service com-

position problems in the various settings. Specifically, we present and discuss the following results.

Data-centric view. Under this view, we address the composition problem in two distinct settings: one in which the views can be used freely (*unrestricted access*), and one in which views have an input-output access pattern, which requires certain arguments to be given as input to the query (*restricted access*). For the unrestricted access setting, we essentially recast the results on view-based query answering for regular path queries (Calvanese et al. 2000c; 2000a; 2000b; 2002) in the context of open APIs. The main difference being that since APIs are implemented through concrete programs, which need to be deterministic, we consider regular path queries expressed as DFAs. We discuss the result that computing the certain answers to a query based only on the available APIs is intractable, and in particular CONP-complete in data complexity, i.e., with respect to the size of the data stored in the views represented by the APIs. To overcome this high data complexity, we introduce the notion of rewriting, and show that using this notion, we can build a DFA that, when evaluated over the view extensions, computes an approximation of the answers in polynomial time. We also present a new formulation of the rewriting in terms of Datalog (a logical language for querying deductive databases), rather than in terms of a DFA, by which we can compute the set of answers to the target query with a more efficient method.

For the restricted access setting, we present the first investigation on the service composition problem. Based on its relationship with constraint satisfaction discussed in (Calvanese et al. 2000c), we again show that computing the certain answers to a query based only on the available APIs with restrcited access is CONP-complete in data complexity. As in the case of unrestricted case, we also study a rewriting-based approach, by which we can build a DFA that is able to compute an approximation of the answers in polynomial time. And, again, we show that from the rewriting we can generate a Datalog program that computes the result of the rewriting more efficiently than the corresponding DFA.

Process-centric view. Under this view, there are two fundamental ways to achieve a composition, which we call *static* and *dynamic*, respectively. The aim of the former is to find the DFA such that each sequence of actions it denotes is guaranteed to be consistent with the target service despite the nondeterminism of the domain, i.e., despite the uncertainty in the initial condition and the nondeterministic effects of actions. We call this form of composition simply *composition*, and we observe that it resembles the notion of *conformant planning* in AI Planning(Rintanen 2004), i.e., finding a sequence that, in spite of partial knowledge on the outcome of the actions themselves, is guaranteed to satisfy the goal property. Instead, composition in the dynamic setting, which we call here *orchestration* (Berardi et al. 2003; Balbiani, Cheikh, and Feuillade 2009; Bouguettaya, Sheng, and Daniel 2014), aims at finding the DFA that is able to orchestrate the available services dynamically, meaning that, at each step, the DFA takes the sequence of actions executed so far into account, and uses this knowledge to decide the next action in a way to guarantee the coherence with the target service. Orchestration resembles the notion of *conditional planning* (Rintanen 2004) in AI Planning, i.e., finding a plan for satisfying the goal property that at each step prescribes the action to do conditionally, based on the outcome of the actions at the previous steps. We address the composition problem in both the static and the dynamic settings. In the static one, the key observation is that we can recast this problem as a regular-language-rewriting problem, where we are trying to rewrite the regular language corresponding to the target service in terms of a new DFA, whose atomic actions are the available APIs. From this observation, we derive the result that the composition problem in this setting is PSPACE-complete. In the dynamic setting, we solve orchestration by devising a specific game between a client, corresponding to the orchestrator, and an adversary, which is able to call the available APIs. In particular, we reformulate the game as an infinite-duration turn-based game, and we show that orchestration can be solved in polynomial time.

The paper is organized as follows. In Section 2, we review the technique described in (Calvanese et al. 2002) for rewriting regular languages. This technique is then used extensively in the subsequent sections. Section 3 illustrates the problems and the techniques for service composition in the data-centric setting, by addressing both the case of unrestricted and the case of restrcited access. Section 4 deals with the process-centric setting, and presents techniques for both composition and orchestration. Section 5 concludes the paper with a discussion on future work.

2 Rewriting of Regular Languages

In this section, we review a technique from (Calvanese et al. 2002) for the following *regular language rewriting problem*: Given a regular language E_0 and a finite set $\mathcal{E} = \{E_1, \ldots, E_k\}$ of regular languages expressed over a *sources alphabet* Σ, re-express, if possible, E_0 by a suitable combination of $E_1, \ldots, E_k$. We assume that associated with $\mathcal{E}$ we always have a *target alphabet* $\Sigma_{\mathcal{E}}$ containing exactly one symbol for each regular language in $\mathcal{E}$. For a symbol $e \in \Sigma_{\mathcal{E}}$, we call *expansion of* e *wrt* $\mathcal{E}$, denoted $exp_{\Sigma}(e)$, the regular language over the source alphabet Σ associated with e. We extend the notion of expansion in the natural way to words and languages over $\Sigma_{\mathcal{E}}$, i.e., for a word $e_1 \cdots e_m \in \Sigma_{\mathcal{E}}^*$, we have that

$$exp_{\Sigma}(e_1 \cdots e_m) = \{w_1 \cdots w_m \mid w_i \in exp_{\Sigma}(e_i), \text{ for } 1 \leq i \leq m\}$$

and for a language $L \subseteq \Sigma_{\mathcal{E}}^*$, we have that

$$exp_{\Sigma}(L) = \bigcup_{w \in L} exp_{\Sigma}(w).$$

Thus, $exp_{\Sigma}(L)$ denotes all the words over the source alphabet Σ obtained from a word $e_1 \cdots e_m \in L$ by substituting for each e_i all words of the regular language associated with e_i.

Definition 1 Let R be a language over the target alphabet $\Sigma_{\mathcal{E}}$. We say that R is a *rewriting* of a regular language E_0 over the source alphabet Σ with respect to a set $\mathcal{E}$ of regular languages over Σ if $exp_{\Sigma}(R) \subseteq E_0$. ∎

We are interested also in maximal rewritings, i.e., rewritings that capture in the best possible way the language defined by the original regular language E_0.

Definition 2 A rewriting R of E_0 wrt $\mathcal{E}$ is Σ-*maximal* if for each rewriting R' of E_0 wrt $\mathcal{E}$ we have that $exp_\Sigma(R') \subseteq exp_\Sigma(R)$. A rewriting R of E_0 wrt $\mathcal{E}$ is $\Sigma_\mathcal{E}$-*maximal* if for each rewriting R' of E_0 wrt $\mathcal{E}$ we have that $R' \subseteq R$. ∎

Intuitively, when considering Σ-maximal rewritings we look at the languages obtained after substituting each symbol in the rewriting by the corresponding regular language over the source alphabet Σ, whereas when considering $\Sigma_\mathcal{E}$-maximal rewritings we look at the languages over the target alphabet $\Sigma_\mathcal{E}$. Observe that by definition all Σ-maximal rewritings define the same language (similarly for $\Sigma_\mathcal{E}$-maximal rewritings), and that not all Σ-maximal rewritings are $\Sigma_\mathcal{E}$-maximal. However, $\Sigma_\mathcal{E}$-maximality is a sufficient condition for Σ-maximality, as shown in (Calvanese et al. 2002).

Given E_0 and $\mathcal{E}$, we are interested in deriving a Σ-maximal rewriting of E_0 wrt $\mathcal{E}$. It is shown in (Calvanese et al. 2002) that such a maximal rewriting always exists (although it may be empty). In fact, there is a method that, given E_0 and $\mathcal{E}$ constructs a $\Sigma_\mathcal{E}$-maximal rewriting of E_0 wrt $\mathcal{E}$, which is also Σ-maximal. The method presented in (Calvanese et al. 2002) is based on the idea of characterizing by means of an automaton, which we call A', exactly those $\Sigma_\mathcal{E}$-words that are *not* a rewriting of E_0 wrt $\mathcal{E}$. Observe that a $\Sigma_\mathcal{E}$-word $e_1 \cdots e_m$ is not in a rewriting of E_0 wrt $\mathcal{E}$ if there is a Σ-word in its expansion that is not in E_0. If we can build such an automaton A', then its complement is the maximal rewriting we are looking for, in the sense that it accepts exactly those $\Sigma_\mathcal{E}$-words whose expansions are contained in E_0. The crucial point is the construction of A'.

In this paper, we assume that the set of signatures of the available APIs constitutes the target alphabet. Each API program is constituted by a regular language over the source alphabet. Since such program represents executable code, the associated regular language is in fact given in terms of a *deterministic* finite-state automaton (DFA). Hence, we start from a DFA A_0 for E_0 and let A' have the same states as A_0. With regard to the transitions of A', we place in A' a $\Sigma_\mathcal{E}$-edge e between two states s_i and s_j if there is a Σ-word in the expansion of e that leads from s_i to s_j in A_0. Now, in A' a $\Sigma_\mathcal{E}$-word $e_1 \cdots e_m$ leads from s to s' if in A_0 there is a sequence of Σ-words $w_1 \cdots w_m$ that leads from s to s'. Hence we should let A' accept only those $\Sigma_\mathcal{E}$-words that lead from the initial state to a state that is non-final for A_0. Notice that the automaton A' is *nondeterministic*! Based on this idea, the construction takes E_0 and $\mathcal{E}$ as input, and returns an automaton $A_{\mathcal{E},E_0}$ built as follows:

1. Start with a DFA $A_0 = (\Sigma, S, s_0, \rho, F)$ such that $\mathcal{L}(A_0) = E_0$.
2. Define the nondeterministic finite automaton (NFA) $A' = (\Sigma_\mathcal{E}, S, s_0, \rho', S - F)$, where $s_j \in \rho'(s_i, e)$ if and only if there exists a word $w \in exp_\Sigma(e)$ such that $s_j \in \rho^*(s_i, w)$ [1]. In other words, A' has the same states as A_0, the same initial state s_0, and as final states all states that are not final in A_0. With regard to the transitions, A' has a transition from s_i to s_j labeled with $e \in \Sigma_\mathcal{E}$ if and only if there is a Σ-word in the expansion of e that leads from s_i to s_j in A_d.
3. $A_{\mathcal{E},E_0} = \overline{A'}$, i.e., $A_{\mathcal{E},E_0}$ is the complement of A', which is a DFA.

[1] ρ^* denotes the extension of the transition function ρ to words, defined in the standard way for finite automata (Hopcroft and Ullman 1979).

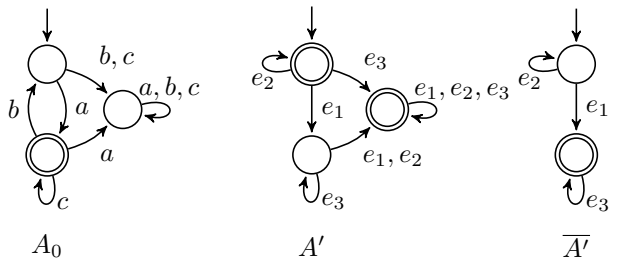

Figure 1: Construction of the rewriting of $a{\cdot}(b{\cdot}a + c)^*$ with respect to $\{a,\ a{\cdot}c^*{\cdot}b,\ c\}$

Step 2 of the construction requires to check whether there exists a word $w \in exp_\Sigma(e)$ such that $s_j \in \rho^*(s_i, w)$. To do so, we consider the automaton $A_0^{i,j} = (\Sigma, S, s_i, \rho, \{s_j\})$, obtained from A_0 by suitably changing the initial and final states, and check the product automaton between $A_0^{i,j}$ and an automaton for $exp_\Sigma(e)$ for non-emptiness. This can be done in nondeterministic logspace in the size of $A_0^{i,j}$ and the automaton for $exp_\Sigma(e)$. The above construction is *correct* (Calvanese et al. 2002), in the sense that: $\mathcal{L}(A_{\mathcal{E},E_0})$ is a $\Sigma_\mathcal{E}$-maximal rewriting of E_0 wrt $\mathcal{E}$. We illustrate the construction by means of an example.

Example 1 Let $E_0 = a{\cdot}(b{\cdot}a + c)^*$, $\mathcal{E} = \{a, a{\cdot}c^*{\cdot}b, c\}$, and $\Sigma_\mathcal{E} = \{e_1, e_2, e_3\}$, with $\mathcal{L}(e_1) = a$, $\mathcal{L}(e_2) = a{\cdot}c^*{\cdot}b$, and $\mathcal{L}(e_3) = c$. The DFA A_0 shown in Figure 1 accepts E_0, while A' is the corresponding automaton constructed in Step 2 of the rewriting algorithm. Since in this example A' is deterministic, by simply exchanging final and nonfinal states we obtain its complement $\overline{A'}$, which is the automaton $A_{\mathcal{E},E_0}$ computed by the algorithm. ∎

It remains to analyze the computational complexity of this technique. The complexity analysis in (Calvanese et al. 2002) is with respect to regular languages represented in terms of regular expressions. Considering that here we start from regular languages represented as DFAs, we can avoid the initial determinization step in (Calvanese et al. 2002), and reduce the the complexity by an exponential, as shown by the next theorem.

Theorem 1 *Let E_0 be represented through a DFA A_0 and let the regular languages in $\mathcal{E}$ be represented through DFAs $A_1, \ldots, A_k$. Let further $A_{\mathcal{E},E_0}$ be the DFA accepting the maximal rewriting of E_0 with respect to $\mathcal{E}$.*

1. The size of $A_{\mathcal{E},E_0}$ is exponential in the size of A_0.

2. *Checking if $A_{\mathcal{E},E_0}$ is nonempty is* PSPACE*-complete with respect to A_0, and is in* NLOGSPACE *with respect to $A_1, \ldots, A_k$.*

Proof. Note that the state set of A' is the same as that of A_0, but while A_0 is a DFA, A' is an NFA. Note also that the size of $A_1, \ldots, A_k$ does not affect the number of states of A'. Finally, $A_{\mathcal{E},E_0}$ is obtained by complementation, which involves a *subset construction* (Hopcroft and Ullman 1979), resulting in an exponential blow-up.

Checking that $A_{\mathcal{E},E_0}$ is nonempty can be done without constructing the automaton in full. Instead, we search for a path from initial to accepting states. Each state of $A_{\mathcal{E},E_0}$ is a set of states of A_0; thus, the search can be conducted in polynomial space with respect to A_0. Constructing the transition relation ρ' of A' requires a search for paths from initial to accepting states of the automata $A_1, \ldots, A_k$. This search can be conducted in nonlogarithmic space with respect to those automata.

To prove PSPACE-hardness with respect to A_0, we reduce from the linear-space bounded nonemptiness of Turing machines. Let M be a Turing machine and $n > 0$ an integer represented in unary. The linear-space bounded nonemptiness problem is to decide if M accepts the empty word using space n. Let Γ be the configuration alphabet of M; that is, each configuration is a word of length n over Γ, and an accepting computation of M is a sequence of configurations that starts in an initial configuration and ends in an accepting configuration.

Given a configuration $C = a_1 \cdots a_n$ and a position $1 \leq i \leq n$, the *local neighborhood* of i, denoted $local(C, i)$ is the triple (a_{i-1}, a_i, a_{i+1}), where we take as default $a_0 = a_{n+1} = \#$ as a special symbol in Γ. It is known (cf. (Hopcroft and Ullman 1979)) that there is a binary relation T_M over Γ^3 such that a configuration C_2 follows configuration C_1 if for $1 \leq i \leq n$ we have that $(local(C_1, i), local(C_2, i)) \in T_M$. We say that two such triples of symbols are *locally related*.

Let $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$. For $1 \leq i \leq k$, let γ'_i be the letter $\overline{\gamma_i}$, which is a *marked* version of γ_i. We take $\mathcal{E} = \{E_1, \ldots, E_k\}$, where $E_i = \{\gamma_i, \gamma'_i\}$, is denoted by the symbol e_i. Thus, every candidate rewriting over $\Sigma_{\mathcal{E}} = \{e_1, \ldots, e_k\}$ is a possible computation of M, with the possibility of some of the symbols being marked.

It remains to describe the target language E_0 to ensure that a legal rewriting R over $\Sigma_{\mathcal{E}}$ must be an accepting computation of M.

1. The length of R is a multiple of n.
2. The first configuration in R must be an initial configuration; and the last configuration must be accepting.
3. If there are two marked symbols in R, then the distance between them must be n, and their local neighborhoods must be locally related.

It is easy to express each condition by a DFA of size $O(n)$. Thus, their conjunction can be expressed by a DFA A_0 of size $O(n^3)$. Now $A_{\mathcal{E},E_0}$ is nonempty precisely if M accepts the empty word using space n. ∎

3 Data-centric View

In this section we study data-centric APIs formalized as queries over a graph database.

A *graph database* (or, simply database) is a finite directed graph whose edges are labeled by elements from a given finite alphabet Σ. Each node represents an objects and an edge from object x to object y labeled by r, denoted $r(x, y)$, represents the fact that relation r holds between x and y. *Regular-path queries* (RPQs) are formulated as regular languages over Σ; here we express such languages using DFAs. The *answer $Q^{\mathcal{B}}$ to an RPQ Q over a graph database $\mathcal{B}$* is the set of pairs of nodes connected in $\mathcal{B}$ by a directed path traversing a sequence of edges forming a word in the regular language $\mathcal{L}(Q)$ defined by Q. We assume that the language associated with an RPQ does not contain the empty word. Indeed, if this were the case, then the RPQ would need to return the set of all pairs (a, a), where a is a node in $\mathcal{B}$, which is not meaningful.

Consider now a database $\mathcal{B}$ that is unknown, but that can be accessed through a finite set $\mathcal{V} = \{v_1, \ldots, v_k\}$ of APIs representing RPQs. Each API v_i has an associated *view definition* $def(v_i)$, that is an RPQ over Σ. We consider the APIs to be *sound* (Abiteboul and Duschka 1998; Grahne and Mendelzon 1999), i.e., for each API v_i in general we are given an extension $ext(v_i)$, which is a *subset* of the result obtained by applying the query $def(v_i)$ to the database $\mathcal{B}$.

We distinguish two cases of *API-based query answering*, depending on the form of access provided by APIs:

- We call *unrestricted access* the case where the APIs can be used freely, i.e., for each API v_i, we have complete access to the *extension* $ext(v_i)$.
- We call *restricted access* the case where the APIs can be used only to return the set of nodes connected by the corresponding query to an initial node given as input. That is, given a node a to an API v_i, return the *restricted extension* $rext(v_i, a) = \{b \mid (a, b) \in ext(v_i)\}$.

Suppose now that a user wants to compute an RPQ Q over the database $\mathcal{B}$. Since the database is accessible only through the APIs $\mathcal{V}$, we need to provide the answer to the RPQ by making use of the APIs only, including taking into account the restrictions on the form of access.

Unrestricted Access. In this case, API-based query answering amounts to what has been studied in the literature as *view-based query answering* (Calvanese et al. 2000a; 2002). In particular, the APIs act as views, and the answer to Q we are looking for corresponds to the *certain answers* of Q with respect to $\mathcal{V}$. We call $adom(\mathcal{V})$ the set of nodes occurring in $ext(v_1) \cup \cdots \cup ext(v_k)$. The set $cert(Q, \mathcal{V})$ of certain answers to Q with respect to $\mathcal{V}$ is the set of pairs (c, d) of nodes in $adom(\mathcal{V})$ such that $(c, d) \in Q^{\mathcal{B}}$, for every database $\mathcal{B}$ such that $ext(v_i) \subseteq def(v_i)^{\mathcal{B}}$, for $1 \leq i \leq k$.

To check whether a pair of nodes (c, d) is in $cert(Q, \mathcal{V})$, we can exploit the correspondence with CSP shown in (Calvanese et al. 2000c). We generate the *constraint template* T of Q wrt $\mathcal{V}$ defined as follows. The vocabulary of T is

$\mathcal{V} \cup \{u_0, u_f\}$, where symbols in $\mathcal{V}$ denote binary predicates, and u_0 and u_f denote unary predicates. Let $A_Q = (\Sigma, S, S_0, \rho, F)$ be an automaton[2] for Q. The structure $T = (\Delta^T, \cdot^T)$ is given by:

- $\Delta^T = 2^S$ is the domain of T;
- $(\sigma_1, \sigma_2) \in v_i^T$, for $1 \leq i \leq k$, iff there exists a word $w \in \mathcal{L}(def(v_i))$ such that $\rho(\sigma_1, w) \subseteq \sigma_2$;
- $\sigma \in u_0^T$ iff $S_0 \subseteq \sigma$, and $\sigma \in u_f^T$ iff $\sigma \cap F = \emptyset$.

Above, we use $\rho(\sigma_1, w)$ to denote the set of states in S reachable from some state in σ_1 by following the word w. Observe that T can be constructed in polynomial space in the size of the expressions Q and $def(v_1), \ldots, def(v_k)$. In particular, verifying the existence of a word $w \in \mathcal{L}(def(v_i))$ such that $\rho(\sigma_1, w) \subseteq \sigma_2$ amounts to verifying whether it is not the case that $\mathcal{L}(def(v_i))$ is included in the language accepted by the automaton $(\Sigma, S, \sigma_1, \rho, S \setminus \sigma_2)$.

Further, from the extensions $ext(v_i)$ and two nodes c, d, we can immediately construct a new database I, called *constraint instance* for $\mathcal{V}$ and c, d, as follows:

- $v_i^I = ext(v_i)$, for $1 \leq i \leq k$,
- $u_0^I = c$ and $u_f^I = d$.

Notably $(c, d) \notin cert(Q, \mathcal{V})$ iff there exists a homomorphism from I to T in (Calvanese et al. 2000c).

Moreover, (Calvanese et al. 2000c) provides a complexity characterization for certain answer computation in terms of combined, expression, and data complexity, cf. (Vardi 1982). In our case, expression complexity is measured in the size $|Q|$ of the query Q and the combined size $\sum_{v_i \in \mathcal{V}} |def(v_i)|$ of the view definitions, while data complexity is measured in the combined size $\sum_{v_i \in \mathcal{V}} |ext(v_i)|$ of the view extensions. Note that query and view definitions typically tend to be short, while data size tends to be large. Thus, data complexity is usually the more significant barrier to query evaluation. Now, checking whether $(c, d) \in cert(Q, \mathcal{V})$ is coNP-complete in data complexity (Calvanese et al. 2000a). Hence we get:

Theorem 2 *API-based query answering under unrestricted access is* coNP*-complete in data complexity.*

To overcome such high data complexity, we can compute an approximation of the answers based on exploiting rewritings. We call such problem *unrestricted access API-based query rewriting*. According to such an approach, an RPQ Q over the graph database alphabet is processed by first reformulating Q into an RPQ R_{max}, called *maximal rewriting*, expressed over the API symbols $\mathcal{V}$, and then evaluating R_{max} over the API extensions. Again, this problem has a correspondent in the database literature, where it is called view-based query rewriting. The relationship between view-based query answering and view-based query rewriting is investigated in (Halevy 2001; Calvanese et al. 2000c; Lenzerini 2002; Calvanese et al. 2007).

Let Q be an RPQ over the database alphabet, and let R be an RPQ over the API alphabet $\mathcal{V} = (v_1, \ldots, v_k)$. We say that R is a *rewriting of Q under APIs $\mathcal{V}$*, if for every graph database $\mathcal{B}$ and for every possible extension $\mathcal{D} = (D_1, \ldots, D_k)$ for $\mathcal{V}$ such that $D_i \subseteq def(v_i)^{\mathcal{B}}$, for $1 \leq i \leq k$, we have that $R^{\mathcal{D}} \subseteq Q^{\mathcal{B}}$.

Among the rewritings, we are interested in the maximal ones. An RPQ R_{max} over $\mathcal{V}$ is the *maximal rewriting* of Q under $\mathcal{V}$ if *(i)* R_{max} is a rewriting of Q under $\mathcal{V}$, and *(ii)* for every rewriting R of Q under $\mathcal{V}$, we have that $R^{\mathcal{D}} \subseteq R_{max}^{\mathcal{D}}$, for every graph database $\mathcal{B}$ and for every extension $\mathcal{D} = (D_1, \ldots, D_k)$ for $\mathcal{V}$ such that $D_i \subseteq def(v_i)^{\mathcal{B}}$.

Actually, as shown in (Calvanese et al. 2002), the maximal rewriting of an RPQ Q under APIs $\mathcal{V}$ can be computed by resorting to the maximal language rewriting presented in Section 2.

Theorem 3 *The maximal rewriting of an RPQ Q under APIs $\mathcal{V}$ is the RPQ $A_{\mathcal{E},E_0}$, where $\mathcal{E} = \{def(v_i) \mid v_i \in \mathcal{V}\}$ is the set of API definitions, and $E_0 = Q$.*

With respect to computation complexity we obtain the following result.

Theorem 4 *The maximal unrestricted access API-based query rewriting of an RPQ Q under APIs $\mathcal{V}$ can be computed in exponential time in the size of Q and in polynomial time in the size of $def(v_1), \ldots, def(v_k)$.*

Turning to actually computing the answer to Q under APIs $\mathcal{V}$ through the maximal rewriting R_{max}, we observe that R_{max} is a DFA, and hence could be transformed into a regular expression, to be evaluated by performing joins and transitive closure operations over the API extensions (Calvanese et al. 2002). This comes with the price of a further worst-case exponential blowup of converting a DFA into a regular expression. Here, however, we propose a different query evaluation technique that avoids this blowup by resorting to a *Datalog program* Π_m that is evaluated over an extensional database formed by the API extensions. Datalog is a well studied query language whose prominent feature is the possibility of using recursion in queries. It can also be seen as a fragment of Prolog where we do not allow for nesting of terms, though the evaluation procedures used for it are typically bottom-up instead of top-down. A Datalog program consists of a finite set of rules of the form $P(\vec{x}) \leftarrow \varphi(\vec{x}, \vec{y})$,where $P(\vec{x})$ is an atom with free variables $\vec{x}$, and $\varphi(\vec{x}, \vec{y})$ is a (possibly empty) conjunction of atoms whose variables are in $\vec{x}$ and $\vec{y}$. The variables $\vec{y}$ are implicitly existentially quantified, while the variables $\vec{x}$ occur free and are used to transfer data from $\varphi(\vec{x})$ to $P(\vec{x})$. The data complexity of evaluating Datalog programs is PTIME-complete. We call *linear Datalog* the fragment of Datalog in which we allow for at most one recursive call per rule. Such a fragment is NLOGSPACE-complete in data complexity (Abiteboul, Hull, and Vianu 1995).

Let the maximal rewriting R_{max} be the DFA $(\mathcal{V}, S, s_0, \rho, F)$ over the set of APIs. The Datalog program Π_m contains the following predicates, which are all binary: *(i)* one predicate s for each state $s \in S$, where $s(x, y)$ states that from node x one can reach node y by executing R_{max} and stopping in s; *(ii)* one predicate v for each $v \in \mathcal{V}$, denoting the extension $ext(v)$ of v; and *(iii)* the

[2] In fact, for this construction it is irrelevant whether we start from a deterministic or a nondeterministic automaton for Q.

$$
\begin{array}{rcl}
s_0(x,x) & \leftarrow & v(x,y) \\
 & & \text{for each } v \in \mathcal{V} \text{ such that there exists } s' = \rho(s,v) \\
s'(x,y) & \leftarrow & s(x,z),\ v(z,y) \\
 & & \text{for each } s,\ s',\ v \text{ such that } s' = \rho(s,v) \\
ans(x,y) & \leftarrow & s(x,y) \\
 & & \text{for each } s \in F.
\end{array}
$$

Figure 2: Rules of the Datalog program Π_m for unrestricted access

answer predicate ans. The rules of the program are shown in Figure 2.

Theorem 5 *For each pair (c,d) of nodes, Π_m returns $ans(c,d)$ if and only if R_{max}, when evaluated over the extension of the views in $\mathcal{V}$, returns (c,d).*

Turning to computational complexity, consider that Π_m has at most one recursive call per rule, hence is a linear Datalog program. Now considering the data complexity of linear Datalog, we obtain the following result.

Theorem 6 *Computing the answer to an RPQ Q under unrestricted access APIs $\mathcal{V}$ through the maximal rewriting is in* NLOGSPACE *in data complexity, i.e., in the size of $ext(v_1), \ldots, ext(v_k)$.*

Proof. The claim for data complexity follows from Theorem 5 and the fact that the data complexity of linear Datalog is NLOGSPACE-complete, and hence can be done in polynomial time.

As for expression complexity, it suffices to observe that the Datalog program Π_m is linear in the number of states of the maximal rewriting R_{max}. The claim then follows from Theorem 4. ∎

We get also a characterization of expression complexity by considering the complexity of computing the maximal rewriting R_{max} established in Theorem 4, and observing that Π_m is linear in the number of states of R_{max}. Specifically the complexity is single exponential in the size of Q, and polynomial in the size of $def(v_1), \ldots, def(v_k)$. Notice that in general we expect the size of the query to be much smaller than the size of the API extensions.

Restricted Access. In the case of restricted access, API-based query answering becomes a form of *view-based query answering* in the presence of views with access restrictions (Li and Chang 2001; Deutsch, Ludäscher, and Nash 2007; Benedikt, Bourhis, and Ley 2012). In this case, in the query answering problem, we assume that also the user RPQ is a query with restricted access, in particular it asks for all nodes reachable from a given node via a path in the language of the RPQ.

To check whether a pair (c,d) of nodes is in $cert(Q,\mathcal{V})$, we can again exploit the construction above involving the constraint template. However this time we do not have direct access to the constraint instance, since the APIs do not return directly the extension but need the first argument as input to return the second. Now, given a node c we can compute for each API all pairs formed by c and the result of invoking the API. This gives us a first approximate extension of each API. Then from each of the nodes obtained in this way we can repeat the process getting a larger extension for each API, and so on, until the extension of each API does not increase anymore. Formally, for each API v_i, the *extension of v_i relative to c*, denoted $ext_c(v_i)$, is defined by simultaneous induction on all APIs as follows:

- $(c,x) \in ext_c(v_i)$, for each $x \in rext(v_i,c)$, for $1 \le i \le k$;
- if $(x_1,x_2) \in ext_c(v_j)$ for some j, then, for $1 \le i \le k$:
  - $(x_1,y_1) \in ext_c(v_i)$, for each $y_1 \in rext(v_i,x_1)$, and
  - $(x_2,y_2) \in ext_c(v_i)$, for each $y_2 \in rext(v_i,x_2)$.

Now, from the extensions $ext_c(v_i)$ and two nodes c, d, we can immediately construct a new database I_c, called *constraint instance relative to c* for $\mathcal{V}$ and d, as follows:

- $v_i^I = ext_c(v_i)$, for $1 \le i \le k$,
- $u_0^I = c$ and $u_f^I = d$.

Then we get the following result.

Theorem 7 *Let Q be an RPQ, $\mathcal{V}$ a set of RPQ restricted access APIs, and c, d a pair of nodes. Further, let I_c be the corresponding constraint instance relative to c, and T the corresponding constraint template. Then $(c,d) \notin cert(Q,\mathcal{V})$ iff there exists a homomorphism from I_c to T.*

Proof. For each API $v_i \in \mathcal{V}$, let $ext(v_i)$ be the extension of v_i, and let I be the corresponding constraint instance. To show the claim, it suffices to show that there is a homomorphism from I to T iff there is a homomorphism from I_c to T. Notice that by definition $ext_c(v_i) \subseteq ext(v_i)$. Hence if there is a homomorphism h from I to T, then h restricted to the domain of I_c is a homomorphism from I_c to T. For the other direction, suppose there is a homomorphism h from I_c to T. Then h can be extended to a homomorphism h' from I to T by mapping all nodes x not occurring in I_c to the empty set, which is an element of T. Notice that, in T by construction, we have $(\emptyset,\sigma_2) \in v_i^T$ for every domain element σ_2 of T and for every view v_i. Hence, h' is indeed a homomorphism from I to T. ∎

We can use the this result to obtain an upper bound for the restricted access case that is analogous to the one for the unrestricted access case. In the following theorem, we prove also a matching lower bound.

Theorem 8 *Restricted access API-based query answering is* CONP*-complete in data complexity.*

Proof. For the upper bound, we observe that the constraint instance relative to a node c can be computed in polynomial time in the number of nodes. The claim the follows from Theorem 7, by observing that checking the existence of a homomorphism from a structure U_1 to a structure U_2 can be done in NP in the size of U_1 (Feder and Vardi 1999).

For the lower bound, we provide a reduction from graph 3-colorability (Garey and Johnson 1979). Let $G = (N,E)$ be an undirected graph to be checked for 3-colorability. The alphabet is given by $\Sigma = \{a_{rg}, a_{gr}, a_{rb}, a_{br}, a_{gb}, a_{bg}, a_s, a_e\}$, where a_{rg}, a_{gr}, a_{rb}, a_{br}, a_{gb}, and a_{bg} denote pairs of colors assigned to the

two vertices of a directed edge and a_s and a_e are two additional symbols. We make use of restricted access APIs $\mathcal{V} = \{v_s, v_e, v_G\}$ with $adom(\mathcal{V}) = N \cup \{x_s, x_e\}$, where x_s, x_e are two nodes not in N. The API definitions and extensions are defined as follows:

$$\begin{array}{lcl} def(v_s) & = & a_s \\ def(v_e) & = & a_e \\ def(v_G) & = & a_{rg} + a_{gr} + a_{rb} + a_{br} + a_{gb} + a_{bg} \\ \\ ext(v_s) & = & \{(x_s, x) \mid x \in N\} \\ ext(v_e) & = & \{(x, x_e) \mid x \in N\} \\ ext(v_G) & = & E \cup \{(x, x') \mid (x', x) \in E\} \end{array}$$

Intuitively, v_G represents G given as a symmetric directed graph, while v_s and v_e are used to connect x_s and x_e to all nodes of the graph. The RPQ Q is such that $\mathcal{L}(Q) = a_s \cdot M \cdot a_e$, where

$$M = \bigcup_{\{x,y,z,w\} \in \{r,g,b\},\ x \neq y,\ y \neq z,\ z \neq w} a_{xy} \cdot a_{zw}$$

Intuitively, M describes all paths of length two that contain a pair of mismatched color pairs, e.g., the pair $a_{rg} \cdot a_{rb}$ is mismatched, because a_{rg} denotes an edge from a red node to a green node, so it should be followed by a_{gb} or a_{gr}.

It is easy to see that, if the graph G is 3-colorable, then there is a database $\mathcal{B}$ containing the API extensions (i.e., such that $ext(v) \subseteq def(v)^{\mathcal{B}}$, for $v \in \mathcal{V}$), on which the RPQ corresponding to M, and hence Q, is empty; therefore $(c, d) \notin cert(Q, \mathcal{V})$. Instead, if G is not 3-colorable, then every database $\mathcal{B}$ containing the API extensions will contain a sequence of two a_{xy} edges representing a mismatched color pair. On such database $\mathcal{B}$, the RPQ corresponding to M, and hence Q, is nonempty, and hence $(x_s, x_e) \in cert(Q, \mathcal{V})$. Notice that the latter holds also taking into account that the APIs are restricted access, since *(i)* in $ext(v_s)$, the node x_s is connected to every node of N, *(ii)* $ext(v_G)$ is a symmetric relation, and *(iii)* in $ext(v_e)$, every node of N is connected to x_e. Hence, starting from x_s, one can navigate to x_e following a path in Q. ∎

Again, to overcome the high data complexity of restricted access API-based query answering, we can resort to rewriting. Notice that the maximal rewriting R_{max} of an RPQ Q under APIs $\mathcal{V}$ described above, being an automaton, can be used directly in the presence of restricted access APIs. Indeed the automaton, starts at a node c and progresses by executing transitions, keeping track of the collected nodes. At each state s, given a node x and a transition $s' = \rho(s, v_i)$, the automaton computes the set of successor nodes for x as $rext(v_i, x)$.

Hence, we obtain the same complexity characterization as for the unrestricted access case, with the proviso that this time we are going to explore only the extensions of the APIs relative to the initial node in the user query.

In order to actually compute the answer to Q under APIs $\mathcal{V}$ starting from node c through the maximal rewriting, again we first compute such maximal rewriting R_{max}, which is a DFA $(\mathcal{V}, S, s_0, \rho, F)$ over the set of APIs, and then generate from R_{max} a *Datalog* program Π_m^c. In this case, the program contains the following predicates: *(i)* one unary predicate s for each state $s \in S$, where $s(y)$ states that from the initial node c one can reach node y by executing the DFA R_{max} and stopping in s; *(ii)* one binary predicate v for each $v \in \mathcal{V}$, denoting the extension $ext(v)$ of v; and *(iii)* the unary answer predicate ans. The rules of the program are shown in Figure 3. Notice that Π_m^c is not only linear but also a *monadic Datalog* program (Cosmadakis et al. 1988).

$$\begin{array}{lcll} s_0(c) & & & \\ s'(y) & \leftarrow & s(x),\ v(x, y) & \\ & & & \text{for each } s, s', v \text{ such that } s' = \rho(s, v) \\ ans(y) & \leftarrow & s(y) & \\ & & & \text{for each } s \in F. \end{array}$$

Figure 3: Rules of the Datalog program Π_m^c for restricted access

Theorem 9 *For each node d, Π_m^c returns $ans(d)$ if and only if R_{max}, when evaluated over the extension of the views in $\mathcal{V}$, return (c, d).*

By analogous considerations, we get the same complexity results of Theorem 6 for the restricted access case as well.

Theorem 10 *Computing the answer to an RPQ Q starting from c under restricted access APIs $\mathcal{V}$ through the maximal rewriting is in* NLOGSPACE *in data complexity, i.e., in the size of $ext(v_1), \ldots, ext(v_k)$.*

4 Process-Centric View

In the previous section, we focused on data-centered tasks and APIs, and we used regular languages as specifications for linked data items. In this section we focus on process-centered tasks and APIs, and we use regular languages to specify action languages. More formally, the alphabet Σ is now viewed as a finite set of atomic *actions*, and a regular language over Σ expresses an action-sequence language. Thus, given a task T specified as a DFA accepting the regular language $\mathcal{L}(T)$, the task T is accomplished by performing a sequence $w = a_1 a_2 \cdots a_n$ of actions in Σ, where $w \in \mathcal{L}(T)$. We assume that $\mathcal{L}(T)$ does not contain the empty word, so some action must be taken to accomplish the task. As in the previous section, in this setting the client does not have direct access to the actions in Σ. Rather, the client has access to a set $\mathcal{V} = \{v_1, \ldots, v_k\}$ of APIs, where each API v_i is a regular languages specified by means of a DFA $def(v_i)$ over Σ. The intuition is that each call to the API v_i results in an execution of an action sequence $w_i \in \mathcal{L}(def(v_i))$. (Again, we assume that $\mathcal{L}(def(v_i))$ does not contain the empty word.) The challenge is then for the client to carry out the task T by means of API calls. The term "carry out" can, however, be interpreted in different ways, giving rise to different synthesis problems. We consider two settings, which we call static and dynamic composition.

Static Composition. In the first setting, the client is looking for a "static" sequence that is *composed* from the set $\mathcal{V}$ of APIs. A *conformant composition* consists of a sequence of API calls that is guaranteed to achieve the goal regardless

of the uncertainty about the nondeterministic effects of API calls. (This is essentially *conformant planning*, cf. (Rintanen 2004).) In our setting, a call to an API v_i results in the execution of an *arbitrary* action sequence $w \in \mathcal{L}(def(v_i))$. A sequence $v_{i_1} \cdots v_{i_m}$ of API calls is *conformant* with the task T if every word $w = w_{i_1} \cdots w_{i_m}$, where $w_{i_j} \in \mathcal{L}(def(v_{i_j}))$, for $1 \leq j \leq m$, is in $\mathcal{L}(T)$. The *API composition problem* is to find a sequence $v_{i_1} \cdots v_{i_m}$ that is conformant with T. The key observation is that we can recast this problem as a regular-language rewriting problem, as described in Section 2, where we are trying to rewrite the regular language $\mathcal{L}(T)$ in terms of a set $\mathcal{E} = \{E_1, \ldots, E_k\}$ of regular languages, where $E_i = \mathcal{L}(v_i)$, for $1 \leq i \leq k$. In that terminology, a sequence $v_{i_1} \cdots v_{i_m}$ of API calls is conformant precisely when $\{v_{i_1} \cdots v_{i_m}\}$ is a rewriting of $\mathcal{L}(T)$ with respect to $\mathcal{E}$. Thus, a conformant plan exists iff the maximal rewriting of $\mathcal{L}(T)$ with respect to $\mathcal{E}$ is nonempty.

Theorem 11 *Every word accepted by $A_{\mathcal{E},T}$ is a conformant composition.*

Corollary 12 *The API composition problem is* PSPACE*-complete.*

Dynamic Composition. In the second setting, we look for a "dynamic" plan, where the client need not obtain in advance a conformant composition of API calls. Rather, the client can decide which API call to make dynamically. This can be viewed as an *orchestration* of the APIs, cf., (Bouguettaya, Sheng, and Daniel 2014). This can be thought as a game $G_{\mathcal{V},T}$, called a *regular API game*, between the client and an adversary. In round j of the game, the client calls an API v_{i_j} in $\mathcal{V}$. The adversary then responds by returning a word w_{i_j}, with $w_{i_j} \in \mathcal{L}(def(v_{i_j}))$. The client wins $G_{\mathcal{V},T}$ when the sequence $w_{i_0} w_{i_1} \cdots w_{i_m}$ is in $\mathcal{L}(T)$. The task T is *realizable* with respect to $\mathcal{V}$ if the client has a *winning strategy* in $G_{\mathcal{V},T}$. Deciding if the task is realizable is the *API-orchestration problem*. If the task is realizable, then we wish for an effective representation of this winning strategy. This is the *API-orechestration synthesis problem*.

Example 2 Let $\Sigma = \{a, b, c, d\}$ be the set of atomic actions. Let T be a task with $\mathcal{L}(T) = a \cdot c + b \cdot d$, and $\mathcal{V} = \{v_1, v_2, v_3\}$ APIs with:

$$def(v_1) = a + b \qquad def(v_2) = c \qquad def(v_3) = d$$

It is easy to see that no conformant composition exists in this case. Every conformant composition must start with v_1, but then it cannot continue with either v_2 or v_3. On the other hand, the client has an orchestration strategy: first call v_1, and then if v_1 returns a call v_2, otherwise call v_3. ∎

On the face of it, the regular API game seems a finite game, which can be solved using standard game-solving techniques. But the twist here is that an API may have an infinite set of responses to a client move, when the language of the API is infinite. Thus, to solve the API orchestration and synthesis problems, we reformulate the regular API game as an *infinite-duration* turn-based game between the client and the adversary. At each round, the client either *selects* an API in $\mathcal{V}$, *waits*, which we denote by $\perp$, or signals the end of the play, which we denote by $\top$. (We denote $\mathcal{V} \cup \{\perp, \top\}$ by $\mathcal{V}'$.) The adversary responds by selecting a symbol from $\Sigma' = \Sigma \cup \{\$\}$, where $\$$ is a new symbol, which we use as an *endmarker*. The configuration of the game after each round is a word $w \in (\mathcal{V}' \times \Sigma')^*$. The projection of the configuration on the second components (the sequence of moves of the adversary) is $proj(w)$. We write $strip(proj(w))$ to denote the word in Σ^* that results from deleting all endmarkes in $proj(w)$.

A play is an infinite sequence of rounds in the game. Let us now describe the conditions for the client to win a play in this game, which we refer to as the *symbol-based* regular API game $G^s_{\mathcal{V},T}$.

1. After the adversary makes a $\$$ move, the client must respond with an API call (a symbol in $\mathcal{V}$) or with $\top$ (otherwise, the adversary wins).
2. After the adversary makes a Σ move, the client must respond with $\perp$ (otherwise the adversary wins).
3. Suppose that the client follows the above rules, and consider a round where the adversary makes a $\$$ move and the client has not yet made a $\top$ move. At this point, the configuration of the game is $w = uv$, where the last round in u is $(\perp, \$)$, and $v = m_1 \cdots m_n$, where $m_1 = (v_i, \sigma)$ for some $v_i \in \mathcal{V}$ and $\sigma \in \Sigma$, and $m_n = (\perp, \$)$. If then $strip(proj(v)) \notin \mathcal{L}(def(v_i))$, then the client wins.

   In other words, the adversary must respect the client API calls, and when the client issues an API call v_i, the adversary must respond with a sequence in $\mathcal{L}(def(v_i))$.
4. Suppose that the client follows the above rules, and makes a $\top$ move (which means that the prior move by the adversary was $\$$). The configuration of the game before that last round is w. If $strip(proj(w)) \in \mathcal{L}(T)$, then the task is accomplished and the client wins. Otherwise the adversary wins.
5. Suppose that the client follows the above rules, but never makes a $\top$ move. Then there are two possibilities. First, from some point on the adversary never makes a $\$$ move, but only makes moves in Σ. Then the client wins. If, on the other hand, the adversary makes infinitely many $\$$ moves, then the client must make infinitely many API calls, and the adversary wins.

Proposition 13 *The client has a winning strategy in $G_{\mathcal{V},T}$ iff it has a winning strategy in $G^s_{\mathcal{V},T}$.*

We now show that we can express the winning conditions of plays if $G^s_{\mathcal{V},T}$ by means of a deterministic finite-state automaton $A_{\mathcal{V},T}$ on *infinite words* over the alphabet $\mathcal{V}' \times \Sigma'$. This automaton accepts an infinite play, which is an infinite word in $(\mathcal{V}' \times \Sigma')^\omega$ precisely when the client wins in this play.

Recall that we have a DFA $T = (\Sigma, S_T, s^0_T, \rho^T, F_T)$ to describe the task action language. We also have that the action language of each API v_i is given as a DFA $def(v_i) = (\Sigma, S_i, s^0_i, \rho_i, F_i)$. Without loss of generality we assume that the state sets S_i's are disjoint. Define $S' = \bigcup_{i=1}^n S_i \cup \{\$\}$. We define $A_{\mathcal{V},T} = (\mathcal{V}' \times \Sigma', S, s_0, \rho, F)$ as follows. The state set is $S = (S_T \times S') \cup \{s_0, acc, rej\}$, where acc and rej

are special *accept* and *reject* sink states, respectively. Thus, $\rho(acc, c) = acc$, and $\rho(rej, c) = rej$, for all $c \in \mathcal{V}' \times \Sigma'$. $F \subseteq S$ defines the acceptance condition, discussed below.

We now define the transition function $\rho : S \times \mathcal{V}' \times \Sigma' \to S$. The idea is that $A_{\mathcal{V},T}$ simulates T from the start, to check that when the client signals the end of the game, the resulting action sequence is in L_T. Also, $A_{\mathcal{V},T}$ simulates $def(v_i)$ from every API call v_i to check that the adversary returns an action sequence in $\mathcal{L}(def(v_i))$.

- $\rho(s_0, (\bot, \sigma)) = \rho(s_0, (\top, \sigma)) = rej$, for $\sigma \in \Sigma$: the client must start the game with an API call.
- $\rho(s_0, (v_i, \$)) = acc$: the adversary cannot return an empty action sequence.
- $\rho(s_0, (v_i, \sigma)) = (\rho_T(s_T^0, \sigma), \rho_i(s_i^0, \sigma))$, for $\sigma \in \Sigma$: simulate both T and $\mathcal{U}_i$.
- $\rho((s, t), (\bot, \sigma)) = (\rho_T(s, \sigma), \rho_i(t, \sigma))$, for $s \in S_T$, $t \in S_i$, and $\sigma \in \Sigma$: simulate both T and $def(v_i)$.
- $\rho((s, t), (\top, \sigma)) = \rho((s, t), (v_i, \sigma)) = rej$, for $s \in S_T$, $t \in \bigcup_{i=1}^{n} S_i$, $\sigma \in \Sigma$, and $1 \leq i \leq n$: the client must wait for \$ to issue an API call or signal play end.
- $\rho((s, t), (\bot, \$)) = acc$, for $s \in S_T$, $t \in \bigcup_{i=1}^{n} S_i$, and $t \notin \bigcup_{i=1}^{n} F_i$: the adversary returned an incorrect action sequence.
- $\rho((s, t), (\bot, \$)) = (s, \$)$, for $s \in S_T$ and $t \in \bigcup_{i=1}^{n} F_i$: the client is ready to act.
- $\rho((s, \$), (\bot, \sigma)) = rej$, for $s \in S_T$ and $\sigma \in \Sigma$: the client must act.
- $\rho((s, \$), (\top, \sigma)) = acc$, for $s \in F_T$ and $\sigma \in \Sigma$: task was accomplished.
- $\rho((s, \$), (\top, \sigma)) = rej$, for $s \in S_T$, $s \notin F_T$, and $\sigma \in \Sigma$: task was not accomplished
- $\rho((s, \$), (v_i, \$)) = acc$, for $s \in S_T$: the adversary cannot return an empty action sequence.
- $\rho((s, \$), (v_i, \sigma)) = (\rho_T(s, \sigma), \rho_i(s_i^0, \sigma))$, for $s \in S_T$ and $\sigma \in \Sigma$: simulate both T and $def(v_i)$.

Note that the size of S_T is the product of the size of T and the combined size of the $def(v_i)$'s.

It remains to define the acceptance condition of $A_{\mathcal{V},T}$ on infinite words. We use here the *co-Büchi* conditions, which specifies which states *cannot* be visited infinitely often (Grädel, Thomas, and Wilke 2002). Let $F = (S_T \times \{\$\}) \cup \{rej\}$. The co-Büchi automaton $A_{\mathcal{V},T}$ accepts an input word if it has a run that visits F only finitely often. Since rej is a sink state, this means that rej was not visited at all. It follows that the adversary returned control to the client only finitely many times and the task T was accomplished.

Proposition 14 *A play of $G^s_{\mathcal{V},T}$ is winning for the client iff it is accepted by $A_{\mathcal{V}.T}$.*

Infinite-duration games where winning plays can be defined by means of deterministic co-Büchi are called co-Büchi games (Grädel, Thomas, and Wilke 2002). Such games can be solved in quadratic time in the size of the automaton defining winning plays (Kupferman and Vardi 2001). Thus, we get:

Theorem 15 *The regular API orchestration problem can be solved in polynomial time.*

Furthermore, the co-Büchi game algorithm yields an explicit winning strategy. In our case, for every state $(s, t) \in S$, the algorithm yields a symbol $\alpha_{(s,t)} \in \mathcal{V}'$ such that the client can win $G^s_{\mathcal{V},T}$ by selecting $\alpha_{\rho(s_0,w)}$ when the game is in configuration w. This can be easily transformed into a strategy in the game $G_{\mathcal{V},T}$. Thus, we get:

Theorem 16 *The regular API orchestration synthesis problem can be solved in polynomial time.*

We can now contrast the static and dynamic settings for accomplishing regular tasks by means of regular APIs. In the static setting we compile the task into a conformant composition, while in the dynamic setting we compile the task into an orchestrator. The computational-complexity price we pay for the static setting is PSPACE-completeness, in contrast to polynomial complexity in the dynamic setting.

5 Conclusions

In this paper, we studied the composition open APIs from a foundational perspective, where services provided by APIs are specified by means of regular languages. We demonstrated the richness and diversity of the composition problem by illustrating several relevant scenarios in both the data-centric and the process-centric settings. For each scenario, we presented techniques and computational complexity results, thus providing a comphrensive picture of the problem.

In the future, we plan to continue our work along several directions. In particular, we plan to investigate the composition problem in the case where the semantics of services behind the APIs is expressed using formalisms that are more expressive than regular languages. For example, in the data-centric setting, it would be interesting to consider a language allowing the edge of the graph database to be traversed both forward and backward, or enabling the expression of conjunctions of regular path queries. While these extensions have already been considered in the case of unrestricted access (Calvanese et al. 2007), at least partially, nothing is known about the composition problem in the case of restricted access. In the process-centric setting, for example, we will consider mechanisms allowing modeling services that export information about the internal state of the computation (conversational services), thus going beyond the stateless nature of traditional APIs. Such mechanisms have been considered, for instance, in (Berardi et al. 2005b; De Giacomo, Patrizi, and Sardiña 2013; Lustig and Vardi 2013). Another important extension to consider is the case where the service modeling language allows one to combine the data-centric and the process-centric views, in the spirit of the artifact-centric approach to process modeling (Nigam and Caswell 2003; Berardi et al. 2005a; Sardiña and De Giacomo 2009; Calvanese, De Giacomo, and Montali 2013; Calvanese et al. 2013; Belardinelli, Lomuscio, and Patrizi 2014). In this case, the main challenge is to design the formalism taking into account the trade-off between expressivity and decidability of the composition problem.

References

Abiteboul, S., and Duschka, O. 1998. Complexity of answering queries using materialized views. In *Proc. of PODS*, 254–265.

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison Wesley Publ. Co.

Balbiani, P.; Cheikh, F.; and Feuillade, G. 2009. Algorithms and complexity of automata synthesis by asynhcronous orchestration with applications to web services composition. *ENTCS* 229:3–18.

Bartalos, P., and Bieliková, M. 2011. Automatic dynamic web service composition: A survey and problem formalization. *Computing and Informatics* 30(4):793–827.

Belardinelli, F.; Lomuscio, A.; and Patrizi, F. 2014. Verification of agent-based artifact systems. *JAIR* 51:333–376.

Benedikt, M.; Bourhis, P.; and Ley, C. 2012. Querying schemas with access restrictions. *PVLDB* 5(7):634–645.

Benslimane, D.; Dustdar, S.; and Sheth, A. 2008. Services mashups: The new generation of web applications. *IEEE Internet Computing* 12(5):13–15.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2003. Automatic composition of e-services that export their behavior. In *Proc. of ICSOC*, volume 2910 of *LNCS*, 43–58. Springer.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Hull, R.; and Mecella, M. 2005a. Automatic composition of transition-based Semantic Web services with messaging. In *Proc. of VLDB*, 613–624.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Mecella, M. 2005b. Automatic service composition based on behavioral descriptions. *Int. J. of Cooperative Information Systems* 14(4):333–376.

Bouguettaya, A.; Sheng, Q. Z.; and Daniel, F., eds. 2014. *Web Services Foundations*. Springer.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Vardi, M. Y. 2000a. Answering regular path queries using views. In *Proc. of ICDE*, 389–398.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Vardi, M. Y. 2000b. Containment of conjunctive regular path queries with inverse. In *Proc. of KR*, 176–185.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Vardi, M. Y. 2000c. View-based query processing and constraint satisfaction. In *Proc. of LICS*, 361–371.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Vardi, M. Y. 2002. Rewriting of regular expressions and regular path queries. *JCSS* 64(3):443–465.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; and Vardi, M. Y. 2007. View-based query processing: On the relationship between rewriting, answering and losslessness. *TCS* 371(3):169–182.

Calvanese, D.; De Giacomo, G.; Montali, M.; and Patrizi, F. 2013. Verification and synthesis in description logic based dynamic systems. In *Proc. of RR*, volume 7994 of *LNCS*, 50–64. Springer.

Calvanese, D.; De Giacomo, G.; and Montali, M. 2013. Foundations of data aware process analysis: A database theory perspective. In *Proc. of PODS*, 1–12.

Cosmadakis, S. S.; Gaifman, H.; Kanellakis, P. C.; and Vardi, M. Y. 1988. Decidable optimization problems for database logic programs. In *Proc. of STOC*, 477–490.

De Giacomo, G.; Patrizi, F.; and Sardiña, S. 2013. Automatic behavior composition synthesis. *AIJ* 196:106–142.

Deutsch, A.; Ludäscher, B.; and Nash, A. 2007. Rewriting queries using views with access patterns under integrity constraints. *TCS* 371(3):200–226.

Feder, T., and Vardi, M. Y. 1999. The computational structure of monotone monadic SNP and constraint satisfaction. *SIAM J. on Computing* 28:57–104.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability — A guide to NP-completeness*. San Francisco (CA, USA): W. H. Freeman and Company.

Grädel, E.; Thomas, W.; and Wilke, T., eds. 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer. Outcome of a Dagstuhl seminar in February 2001.

Grahne, G., and Mendelzon, A. O. 1999. Tableau techniques for querying information sources through global schemas. In *Proc. of ICDT*, volume 1540 of *LNCS*, 332–347. Springer.

Halevy, A. Y. 2001. Answering queries using views: A survey. *VLDBJ* 10(4):270–294.

Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publ. Co.

Hull, R. 2005. Web services composition: A story of models, automata, and logics. In *Proc. of ICWS*, 30–31.

Kupferman, O., and Vardi, M. Y. 2001. Weak alternating automata are not that weak. *ACM TOCL* 2(3):408–429.

Lenzerini, M. 2002. Data integration: A theoretical perspective. In *Proc. of PODS*, 233–246.

Li, C., and Chang, E. 2001. Answering queries with useful bindings. *ACM TODS* 26(3):313–343.

Lustig, Y., and Vardi, M. Y. 2013. Synthesis from component libraries. *Int. J. on Software Tools for Technology Transfer* 15(5–6):603–618.

Navathe, S.; Elmasri, R.; and Larson, J. 1986. Integrating user views in database design. *Computer* 19(1):50–62.

Nigam, A., and Caswell, N. S. 2003. Business artifacts: An approach to operational specification. *IBM Systems J.* 42(3):428–445.

Rintanen, J. 2004. Complexity of planning with partial observability. In *Proc. of ICAPS*, 345–354.

Sardiña, S., and De Giacomo, G. 2009. Composition of ConGolog programs. In *Proc. of IJCAI*, 904–910.

Vardi, M. Y. 1982. The complexity of relational query languages. In *Proc. of STOC*, 137–146.