Integrating Discovery and Automated Composition: from Semantic Requirements to Executable Code

Piergiorgio Bertoli[1] Joerg Hoffmann[2] Freddy Lécué[3] Marco Pistore[1]
[1]: *ITC-Irst - Trento, Italy* e-mail: `[bertoli,pistore]@itc.it`
[2]: *DERI - Innsbruck - Austria* e-mail: `joerg.hoffmann@deri.org`
[3]: *France Telecom - Cesson Sévigné - France* e-mail: `freddy.lecue@orange-ftgroup.com`

Abstract

Web services are conveniently advertised and published based on (stateless) functional *descriptions, while they are usually realized as (stateful)* processes*. Therefore, the automated enactment of complex web services on the basis of pre-existing ones requires the ability to handle services described at very different abstraction levels. This is the main reason behind the current lack of approaches capable to perform automated end-to-end composition, starting from semantic requirements to obtain executable orchestrations of stateful processes. In this paper we achieve such a challenging goal, by modularly integrating a range of incrementally more complex techniques that cover the necessary discovery and composition phases. By gradually bridging the gap between the high-level requirements and the concrete realization of services, our architecture manages sensibly the complexity of the problem: incrementally more complex techniques are provided with incrementally more focused input. The tests of our architecture on a deployed scenario witness the functionality of the platform and its integrability with standard service engines.*

1 Introduction

A key to the success of a distributed business service model stands in the possibility of reusing the wealth of existing services to build new ones, providing novel and complex functionalities. However, given the requirements for a new service, the task of identifying the existing ones useful to its realization, and then combining them, is extremely demanding and time-consuming for a human operator. Therefore, the development of *automated* support tools in this respect is necessary to enact a distributed service view. Ideally, such tools should be able to automatically generate a composed service, starting just from properly formulated user requirements. However, up to now, the search for such a Holy Grail has been unsuccessful. This is mainly due to the fact that services are conveniently advertised and published as (stateless) *functions*, but they need to be implemented as (stateful) *processes*. These representations greatly differ in the degree of detail at which services are described, and in the languages supporting them. Consequently, techniques developed for handling functionally described services differ substantially from those handling their process counterparts. This means that, in order to obtain an end-to-end architecture that discovers published functions and orchestrates their process implementations, we need to chain different, phase-specific techniques. For instance, we might need to understand that some information service might be useful to book a trip just from looking at its advertised I/O, but then, to actually make use of it, we must understand each detail of the protocol behavior that is realized when interacting with it.

Since the complexity of such techniques grows significantly with the degree of detail at which services are considered, phases high in the chain must appropriately and effectively filter out useless services so that phases lower in the chain can focus on sets of services as small as possible. This requires negotiating as gradually as possible the vast complexity gap between the first discovery phases and the late composition phases. This calls for a careful adaptation and tuning of existing techniques, and for the design of novel ones that act as intermediate bridging steps when the gap between existing techniques is too high.

Our contribution in this work is an architecture designed and realized within the context of the Knowledge Web project (`http://knowledgeweb.semanticweb.org/`), performing end-to-end discovery and composition by seamlessly integrating discovery, functional-level composition, and process-level composition. This integration poses serious challenges, in particular at the interface of functional-level and process-level composition, where a complex goal requirement (a "data net") must be generated to guide process-level composition. For the individual components, we utilize state-of-the-art techniques, except for functional-level composition where we provide a novel, advanced technique, able to deal with partial matches.

The current instantiation of our architecture adopts two well-known standard languages, WSMO [4] and BPEL4WS [2], to express requirements and service workflows respectively. This makes our architecture immediately usable in conjunction with standard web services engines such as Active Web Flow, as witnessed by our experiments on a trip booking scenario.

In Section 2, we use a reference "trip booking" scenario to clarify the roles of the various representation levels, and associated languages. In Section 3, we describe in detail our architecture. In Sections 4, 5, 6, 7 we describe each of its modules, implementing service discovery, semantic reasoning on ontological data, and functional and process level service composition respectively. Finally, in Section 8 and Section 9, we describe our tests on the reference scenario, and discuss related work and future steps.

2 A Reference Scenario

In essence, a service is a program which provides a web-accessible functionality, and which can operate and evolve autonomously from other services. Services can be published over the net, searched, and composed; for these purposes, several description languages have been proposed, following one of two main streams. Semantic description languages such as WSMO and OWL [4, 12] refer to ontologically defined concepts and, in general, focus on a functional view of services in terms of interfaces. Such semantic descriptions are very handy for discovery purposes, but make it difficult to define structured workflows like those presented by actual service implementations. These are the focus of behavioral languages such as BPEL4WS [2], which view services as stateful protocols, but lack any reference to semantics. Some attempts, based on annotating BPEL4WS models [14], or on adopting the so-called OWL process model, aim at integrating these two different views.

In order to clarify the roles of the various representation styles and languages in our framework, we consider a reference scenario where the employee of a firm must set up for a work trip. To do so, he must present a ticket request to the employer's administration, complete with departure date and time, cost, and ticket details. He can gather these data by looking for some on-line information service, and then interacting with them. The administration will either accept or refuse such a proposal, and in the former case, it will provide the employee with some means to pay the ticket, e.g. a credit card number, or a draft cheque. The employee will use it to finalize the trip set-up by buying the ticket via some payment service, and by finally providing a receipt to the administration.

We intend to automate the procedure by locating and orchestrating existing relevant services. In particular, we assume that an ADMIN service is exposed representing the administration, that an INFOTRAINS and an INFOFLIGHT services are published to provide ticket informations over international train/flight routes, and that a BUYCARD and a BUYCHEQUE services are exposed to pay with credit cards or cheques respectively. If we restrict our attention to such means of transportations and payment, we can automate the process by orchestrating these five services. Moreover, to add realism, our scenario includes other services, some of which might seem at first sight relevant, such as e.g. a VIAMICHELIN service that provides itinerary information for different locations, and some local information services such as INFOITALY and INFOFRANCE that only provide informations for trips within specific countries.

Services like the above can be conveniently advertised and searched by making use of functional WSMO descriptions, also called *capabilities*. A capability declares the I/O requirements of a service in terms of concepts defined within a (WSMO) ontology. For instance, the one below is the WSMO description of the ADMIN service, abridged from some low-level details for readability purposes.

```
webService      _"http://www.sra.itc.it/KW/Admin"
importsOntology _"http://www.sra.itc.it/KW/TripOntology.wsml"
capability      _"http://www.sra.itc.it/KW/Admin#AdminCap"

sharedVariables {?tiItin, ?tiPrice, ?trItin}
precondition definedBy
  ?trip[itin hasValue ?trItin ] memberOf Trip
  and
  ?ticket[itin hasValue ?tiItin, price hasValue ?tiPrice]
     memberOf Ticket.
postcondition definedBy
  ?auth memberOf PaymentAuth.
assumption definedBy
  ?ticket [itin hasValue ?trItin]
  and
  exists ?auth
    (?auth [price hasValue ?tiPrice] memberOf PaymentAuth).
effect definedBy
  ?auth [price hasValue ?tiPrice] memberOf PaymentAuth.
```

Such WSMO description refers to an external ontology `TripOntology.wsml`, which defines the relevant concepts `Trip`, `Ticket` and `PaymentAuth`. The `precondition` and `postcondition` parts define the I/O signature of the service, i.e. a payment authorization is produced in response to a trip and ticket request. The `assumption` states that the service succeeds if the ticket complies with the trip, and a suitable payment authorization exists. The `effect` specifies that the payment amounts to the ticket price.

High-level requirements are specified as WSMO *goals*, which are analogous to capabilities. For instance, in our scenario, the high-level goal of booking a ticket for the trip we have in mind is realized as follows:

```
goal _"http://www.sra.itc.it/KW/Goal"
importsOntology _"http://www.sra.itc.it/KW/TripOntology.wsml"
capability      _"http://www.sra.itc.it/KW/Goal#GoalCap"
sharedVariables {?Itin}
precondition  definedBy
  ?trip         [itin hasValue ?Itin] memberOf Trip.
postcondition definedBy
  ?bookedTicket memberOf BookedTicket.
effect        definedBy
  ?bookedTicket [itin hasValue ?Itin] memberOf BookedTicket.
```

In the remainder of the paper, we use an IOPE (inputs, outputs, preconditions, effects) style terminology for web service capabilities, to better relate to standard concepts in technologies used by PLC and FLC. Here, the inputs and outputs correspond to (the respective parts of) the WSMO precondition and postcondition; preconditions and effects correspond to WSMO assumption and effect.

The realization of services, however, may not consist of a capability declaration, since services, in general, are implemented by stateful, multi-phase protocols. For instance, in our scenario, the ADMIN service first requires in input the trip details and those of the proposed ticket, and decides whether the proposal is viable. If not, it answers with a refusal; otherwise, it chooses which payment means to use, and provides it to the requester. Then, it suspends, waiting for either a receipt, or a disconfirmation of the acquisition of the ticket, before terminating. One common choice to implement such protocols is by BPEL4WS specifications, complemented by WSDL interfaces. The WSDL specification for ADMIN describes the input/output ports and operations by which the service can be invoked or reply. The BPEL4WS part specifies in detail the process flow: first, two RECEIVE constructs are sequenced to get the trip and ticket details, then a SWITCH construct models the approval/refusal of the request, and so on. Following [14], we assume that the link between semantic WSMO service specifications and their BPEL4WS counterparts is given in terms of simple semantic annotations of BPEL4WS activities. These identify the input and output concepts, their relationships, and attach logical values to the final service states. This way, it is possible e.g. to declare that some semantic relationships hold at some stage, and to establish the logical status of the service:

```
<receive operation="requestAuth" variable="req"
   semann="/req/itin : Itinerary, req/cost : Price"/>
...
...
<onMessage operation="receipt" variable="rec"
   semann="/rec/cost : Price,
   /rec/cost = /req/cost, /Admin/status = success"/>
```

3 Architecture

A conceptual view of our modular architecture is represented in Fig. 1, and shows how end-to-end composition of web services is realized in a fully automated way, starting from semantic requirements, to obtain a stateful service which can be deployed and executed to orchestrate existing, and automatically discovered, component services. While in the following we describe our architecture instantiation supporting semantic WSMO requirements and BPEL4WS implementations of services, we stress that the architecture design is language-independent; different instanitations are possible, and in our agenda.

In the architecture, user-provided semantic requirements are the starting input to exploit discovery techniques that search databases of I/O service capabilities. In a nutshell,

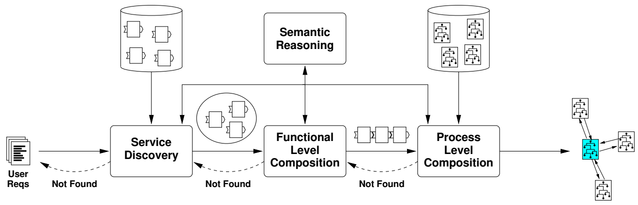

Figure 1. The discovery-composition chain.

the discovery module considers the I/O parts in a database of capabilities to extract a set of services whose I/Os can be linked to cover the I/O of the top-level requirement. Disregarding preconditions and effects greatly simplifies the analysis and allows scaling up to large sets of capabilities; in turn, a relevant part of the interaction taking place amongst services is ignored.

Due to the high level of abstraction considered, the result of Discovery may actually be inadequate, and either contain too many services, or not enough. First, it may include services useless to the goal - such as e.g. the VIAMICHELIN service in our scenario. Second, some of the discovered services may turn out not to be usable, since, in order to work, they require some specific preconditions to hold and these may not be guaranteed. For instance, the INFOITALY service might seem useful; however it only works if the itinerary starts and ends in Italy, making it unusable for our purposes. For these reasons, we need to recur to a capability-based composition to check that the discovered services really cover, at a functional level, the intended goal, and to prune away useless ones. Compositions based on such a functional view of services are known as *functional level compositions* (FLC) [10]; we devise a novel FLC technique that allows partial matches of input and output types, postulating only that the intersection of the involved types is non-empty. The search involved in FLC is sensibly more complex than the one performed during discovery, since FLC requires considering the interaction between the conditions enabled by the application of a service and those required by the services executed later on.

Once FLC extracts a set of services whose capabilities can be combined together to obtain the (capability of) the top-level requirement, we can finally try and combine their actual, stateful implementations. Thus, the annotated BPEL4WS processes (and their WSDL complements) associated to the services identified by FLC are the input to a *process level composition* (PLC) in the style of [11], whose result is a new, executable protocol that interacts with the existing ones to achieve the top-level requirement. Notice that PLC is significantly more complex than FLC, since it is a general form of program synthesis; therefore it is crucial to run it over a restricted number of components services; the contribution of FLC to prune useless discovered services is vital in this respect. Moreover, for PLC, the semantic goal must be recasted to speak of process data and annotations by an appropriate extraction and grounding phase.

A final important remark is that Discovery,FLC and PLC may all fail to find a result; e.g., the processes in input to PLC cannot be composed in a satisfactory way, and similarly for FLC and Discovery. While failure of Discovery amounts to a failure of the overall process, failures of PLC and FLC trigger backtracking in the architecture: e.g. if FLC fails, then Discovery is required to progress, finding a different (and richer) set of services for FLC. Thus, the presence of caching mechanisms is crucial, especially for Discovery and FLC, which may be required to build incrementally upon previous computations.

The architecture is completed by an ontological reasoning module, which provides a vital infrastructural support to Discovery, PLC and FLC. Ontological reasoning is required to map a semantic requirement into a process-level requirement, to allow for partial concept matches during the various phases, and in general for understanding the relationships amongst different semantic concepts. This module provides a set of key functionalities, such as ontology merging and concept entailment checking, and relies on modeling ontologies in terms of a description logics, along the lines of e.g. [12]. The power of such a module is therefore crucial to the performance of the overall architecture.

4 Discovery

The wide-spread standard for service discovery in service-oriented architectures, supported by well-established technologies, is to search a so-called *service registry*, where matching is based purely on the syntax of the inputs and outputs exchanged by the services. The "semantic" alternative to this is to annotate the services with an ontologically defined semantics – e.g. WSMO capabilities – and match the services based on reasoning. This solution, adopted in e.g. [13], is computationally more costly, but has the potential to yield much better results. In our current architecture, we use a third alternative that, in a nutshell, uses the ontology to compile discovery queries posed to a syntactic service registry. We adopt the de-facto standard registry mechanism, Universal Description Discovery and Integration (UDDI), as our instrument for publishing and finding service descriptions; more specifically, we exploit the UDDI **tModel** data type to provide the technical specifications of services.

In our implementation, the ontology structure is used to pre-compile technical specifications of services which describe all of the possible requests their interfaces are meant to satisfy, so that UDDI search can perform a simple (exact) matching on such features to return (URLs pointing to) the WSDL/WSMO/BPEL4WS service descriptions. This amounts to publishing services as **tModel**s that contain sets of capabilities, representing the possible instantiations of concepts into subconcepts that can be manipulated by the service.

UDDI search alone is only the base step of the process implemented within the Discovery module, and needs to be repeatedly invoked until a set of services is collected that are deemed sufficient for the subsequent composition phases. We realize such service collection by means of a breadth-first forward search; at each step, the set of all data-types produced by the services discovered so far is passed as input to the UDDI registry search, so to discover every service that can be possibly activated by combining results of previously discovered services. Service collection stops either when the sets of output datatypes cover those in the output portion of the WSMO requirement, or when no new service is returned by UDDI search, i.e., when a fixpoint is reached.

We remark that the failure of later composition phases may require triggering advances of discovery on the basis of previously cached results. When such re-triggering takes place, a different stopping criteria than type coverage must be used, since type coverage proved not to be sufficient; in our case, we stop after a given number of service collection iterations. At any stage, should service collection reach a fixpoint without satisfying the stopping criteria, the discovery module signals that no composition is possible by returning a top-level failure message.

5 Ontology Reasoning

Ontology reasoning plays a fundamental infrastructural role in our architecture, since each stage in the chain produces and/or receives services described on ontological concepts. Thus, ontology reasoning has to deal with a set of service-specific ontologies $\mathcal{T}_1, \ldots, \mathcal{T}_n$, and with the ontology associated to the input semantic requirement, $\mathcal{T}_\rho$. Detecting the nature of roles and relations of concepts, coming from such different sources, is necessary to pursue a semantically sound functioning of the overall chain. In particular, to enable FLC and PLC to deal with significant scenarios, we must check *concept subsumption*, *mutual exclusion*, and *disjunctive coverage* properties. Roughly spoken a concept subsumes another one if the latter is more specific than the former. Two concepts are mutually exclusive (mutex) if they admit a least common subsumer different from $\top$ in an ontology $\mathcal{T}_i$, and have no instance in common. A set of concepts is a disjunctive cover of a concept D if they are pair wise mutex, and D is contained in their union.

Knowledge and reasoning about the above properties is required to infer *partial matches* among input and output parameters of services. For instance, in our scenario, we need to book a ticket for a trip, where a priori it is not known whether the trip is by train or by airplane, and web services are only available for either of the two. From a semantic point of view, the input A of the desired service does not exactly match any of the inputs A_i of the existing services. Instead, one must use partial matches where $A_i \cap A$ is satisfiable, and discover a set of concepts A_i that covers A. Here, subsumption enables to infer if $A_i \cap A$ is satisfiable, while disjunctive cover computes a set of concepts that covers A.

The inference of the knowledge of the domain required by FLC and PLC components is performed by the ontology reasoning module. It first performs a simple 'ontology merging' phase where all ontologies defined in services, and the one associated to the requirement, are parsed, and unioned in a so-called *knowledge base* $\mathcal{KB}_I$. The merging phase relies on a standard *unique name* assumption: a concept is uniquely associated to a name, therefore if the same name appears in two different ontologies, they refer to the same concept. The subsumption relations explicitly stated within $\mathcal{T}_\rho, \mathcal{T}_1, \ldots, \mathcal{T}_n$, are completed using reasoning. The reasoning can be performed by existing tools, e.g. Pellet [16], FaCT++ [8], which adopting HTTP-based DIG protocol (particularly convenient in a modular architecture such as ours). All inferred subsumptions are added into the knowledge base $\mathcal{KB}_I$; this way, at later stages, subsumption and atomicity checks can be performed very effectively.

Once the subsumption inference is in place, mutual exclusion and disjunctive coverage properties are determined. Mutual exclusion is determined by checking consistency of concepts, using the reasoner; we restrict these checks to siblings, i.e., to pairs of concepts with a common subsumer concept. For disjunctive cover properties, we make a standard *closed world assumption*, stating that the set of subsumees of concept A covers A (i.e., that the set of concepts in the knowledge base $\mathcal{KB}_I$ is complete). As the result of the overall process, we obtain an enriched knowledge base $\mathcal{KB}_D$ which can then be easily and effectively queried by the various architecture modules (Fig. 1) by means of its DAG-structured subsumption hierarchy.

6 Functional Level Composition

Once discovery has come up with an approximation of the needed services, and ontology reasoning has come up with the corresponding ontology, functional level composition (FLC) is invoked, to check the consistency of the result and as a further step for decreasing the complexity prior to process level composition. In difference to discovery, FLC considers the capability descriptions in full, taking into account their entire structure instead of just the I/O behavior. FLC builds a solution that is sound relative to these informations. This is performed by a suitable compilation of the composition problem into an AI Planning task. The AI Planning problem is related to that of service composition, and the definition of planning operators is close in spirit to the capability abstraction of web services. Moreover, state-of-the-art AI Planning tools exhibit fine scalability. Thus, it is unsurprising that compiling composition into AI Planning is not a new idea. However, to sensibly deal with scenarios based on semantic ontologies, we need to allow *partial matches*, as explained in Section 5. Doing so requires support for background theories. In the example from Section 5, the background theory must be considered to find out that $A_i \cap A \neq \emptyset$, and to combine a set of A_i that covers A. In our reference scenario, the same difficulty appears when composing a service to book a ticket for a trip, where a priori it is not known whether the trip is by train or by airplane, and web services are only available for either of the two. None of the new scalable planning tools allow background theories. Hence they can model only exact matches, and cannot deal with our scenario. The few approaches that deal with background theories, e.g. [5, 6], compile the problem into general deduction, which is hardly scalable even in intuitively simple domains.

The main obstacle in allowing background theories in planning is that they severely complicate the evaluation of effects. Every time an action effect changes the truth value of a proposition, the state may become inconsistent with the background theory, and must be "revised" accordingly to obtain the possible successor states. This is well known in AI as the "ramification problem". A standard solution is to define the possible successors as all those states that satisfy both the effect and the background theory, and that differ minimally from the previous state. It is beyond the scope of this paper to define and examine this formally; it should be obvious – and is well known in AI – that such a complicated definition of semantics leads to severe computational difficulties for a composition process.

Our key observation in this work is that there is an interesting *special case* of FLC, where the ramification problem is much less severe. Namely, when an operator (a service) generates a *new* constant, often the effect literals just set the basic properties of that new constant (and the background ontology triggers the implied properties). E.g. in our reference scenario the INFOTRAINS service effects are all related to the produced ticket. The crucial point is that such effect literals do not affect the previous state – they refer to a newly introduced constant which previously did not exist. Hence no inconsistencies can arise, and the successor states can be characterized simply by the conjunction of effect and background theory. We adopt this special case, with background theories allowing constraints of the form $\forall \overline{x}.L_1(\overline{x}) \vee \cdots \vee L_k(\overline{x})$. Note that concept hierarchies have such constraints, taking the form $\forall x.(\neg A(x) \vee B(x))$.

Our second observation is that, with a further simplification, we can compile our problem into a standard notion of planning under uncertainty; namely, *start state uncertainty*, in the form of a propositional CNF formula, where a solution has to work for every possible start state satisfiying that formula. We prove that the special case identified above can be compiled into this form of planning, given one additionally pre-fixes a set of "potential" constants (bounding the outputs that can be produced by the services). Our key observation is that, in this setting, the necessary reasoning about the background theory *can be performed once and for* *all at the start state already.*

In more detail, our compilation associates every service output with one or more tuples of potential constants; it remembers for each such constant c whether it has yet to be generated, by means of a proposition $exists(c)$ initialized as false. The proposition $exists(c)$ becomes true as an effect of the respective service, and its negation is a precondition of that service. Also, the proposition is a precondition of every other service instantiation that has c as an input. The intuitively surprising operation is then that the original action effects are all *moved into the start state formula.* Together with the background theory, this yields a propositional CNF that characterizes exactly all possible configurations of the pre-fixed set of constants. During search, the only thing that changes is which constants actually exist.

As an illustrative example, let's consider a small part of our use case. Say we want to book a ticket for a given trip between two locations; the trip may be either a train trip or a flight trip; the two existing services treat either of those. Then the planning task uses the predicates $train, flight, ticket, exists$, and the constants IN, OUT. The start state is $(train(IN) \vee flight(IN)) \wedge ticket(OUT) \wedge \neg exists(OUT)$. The train trip service has the precondition $traintrip(IN) \wedge \neg exists(OUT)$ and the effect $exists(OUT)$; accordingly for the flight trip service. The end state is $exists(OUT)$. The composed solution tries both services in sequence. After executing this sequence, a ticket has been produced in all possible cases.

It is not difficult to prove that our compilation is sound, and complete provided the pre-fixed set of constants is large enough. To solve the compiled tasks, we chose to use CONFORMANT-FF [7], a recent scalable planning tool that uses SAT reasoning to deal with start state uncertainty. We designed a front end for the tool, taking in the result of discovery and ontology reasoning – ontology, available web services, and goal service – and putting out files in the planning language understood by CONFORMANT-FF. The planning task is then solved, and its solution is a sequence of web services that realizes the goal service (at the capability level). Empirically, our experience is that this is quite effective. In all our use case variants, the planning takes only split seconds.

7 Process Level Composition

The PLC module exploits and adapts the state-of-the-art composition technique first defined in [11], based on interpreting (BPEL4WS) protocols as finite state machines, and on considering so-called *datanet* requirements. Such datanet requirements are very different in nature and content from the original WSMO goal, and their automatic generation starting from the semantic WSMO goal and from the result of FLC is far from trivial. This is the one relevant adaptation we had to put in place to link the datanet tool of [11] to our architecture.

In a nutshell, the main idea behind the datanet approach is that a composition goal can be conveniently structured into a *control flow* and a *data flow* sections. The control flow describes the desired final states of the services involved in the composition, and in our case can be specified as a logical formula over semantic BPEL4WS annotations. For instance, ideally, in our scenario, every service involved in the orchestration should terminate in a successful state: the information service has been able to respond, the customer has a ticket for his desired trip, the payment has been carried out successfully, and the administration has been informed. The data flow complements the control flow by identifying the necessary dependencies among data manipulated by different components. For instance, in our scenario, the trip itinerary should be the same for all involved services, and the ticket price proposed by the information service should also be the one received by the ADMIN for approval, and then used while paying.

The data flow represents data dependencies in a modular fashion, and resembles, at a first sight, a combinatorial network, where inputs and outputs of services are connected by by means of "*gates*" which describe the way data are propagated between nodes. A small set of gate types is enough to represent all possible data dependencies. In the data flow depicted in Fig. 2, which is a subset of the data flow for our scenario, FORK gates (1,2,6,7) perform multiplexing of data; MERGE gates (3,4,5) perform demultiplexing of data, and FILTER gates (8,9,10) are used to allow discarding data. The full set of elements is introduced in [11], and given an automata-based semantics. The datanet approach also accounts for the possibility that component services act in an uncontrollable way, according to some (undisclosed) internal logics. For instance, in our scenario, it is impossible to decide a priori whether the BUYCARD or the BUYCHEQUE service will succeed in buying a ticket or not. In these cases, a single top-level desiderata (e.g. *"have all components succeed and have data dependencies satisfied"*) may not always be achievable, and some fall-back course of actions must be provided, so that some minimal requirements are nevertheless satisfied (e.g. *"the* ADMIN *is consistently notified on the result of buying the ticket"*). To handle these situations, datanet requirements are structured in two layers: a preferred "main" part, and a "recovery" part used as the fallback; both layers are split into a control and a data flow parts.

The main issue in making use of the datanet approach in our context stands in the big difference between the original, WSMO-expressed semantic goal, and the datanet goal we need to instantiate for the services returned by the FLC phase. First, the WSMO goal only (implicitly) refers to a nominal interaction with services, corresponding to the "main" layer of the datanet requirement. Second, the WSMO goal, which is given before performing discovery or functional composition, abstracts away completely from the actual components that will take part to the orchestration; on the contrary, each portion of the datanet requirement refers explicitly to (semantic annotations or interfaces) of such component services. As such, the transformation of the WSMO-described composition goal into its datanet counterpart is far from trivial. To perform such transformation, we rely on the fact that the service sequence returned by FLC already identifies, albeit implicitly, control flow requirements (i.e. which services need to succeed) and data flow requirements (i.e. possible ways of passing data). Therefore, the extraction is performed by closely analyzing the capabilities and ordering of the services identified by FLC, also making use of ontological reasoning.

The most complicated step is obtaining the data flow; for this, we first extract a set of "goal paths", each path specifying a possible routing of the data in input, through some component services, so that the finally obtained data fits the output and effect of the WSMO requirement. The extraction of goal paths is performed by a forward search that progresses a set of "available concepts" in every possible way, according to the plan produced by FLC, until the produced concepts match the ones that need to be returned by the composed service. Notice that here, differently from Discovery, preconditions and effects are also considered, to properly compute data relationships between inputs and outputs, and this analysis uses ontology reasoning to allow for partial concept matches. After goal paths are obtained, they are combined into a data flow. Namely, since nodes must appear only once in a data flow, it is necessary to understand in which case does an input node contribute to more outputs, and to insert appropriate FORK gates in those cases. Similarly, when a single data may come from one of several inputs, the appropriate MERGE gates must be built. FILTER gates are added to the inputs of each service, so that the goal is also valid when such a service needs not be invoked. The result of this process is a set of sub-networks like the one of Fig. 2; their union constitutes the required data flow.

The control flow is identified by analyzing the final states of goal paths, extracting a DNF formula, where each disjunct corresponds to a desirable final configuration for the set of involved services. This is performed both for the "main" portion of the datanet (where invoked services are assumed to succeed) and for the fallback (where they are also allowed to fail).

The extracted data and control flow represent the constraints that must be respected by the desired orchestrator. For instance, gates 1,2,4,9 in Fig.1 represent the constraint that *"should the* ADMIN *receive an itinerary, it must be the one generated by either* INFOTRAINS *or* INFOFLIGHT*"*. This is taken in charge by the WSYNTH tool presented in [11], which we integrate: control and data flow specifications are interpreted as automata, and used to drive the search process which extracts the orchestrator by searching all admissible interactions with the component services.

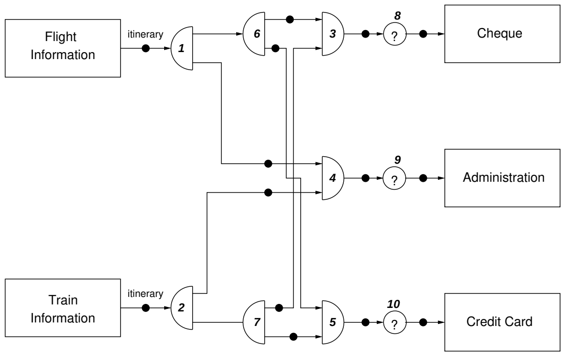

Figure 2. A sub-datanet for the scenario.

8 Experiments

We implemented our architecture, and tested it over our reference scenario. Each module in the architecture relies on powerful state-of-the art technologies and tools to perform the standard tasks. Discovery is based on the jUDDI implementation of the UDDI specification. Ontology reasoning adopts the WSMO4J implementation for WSMO parsing, and makes use of the FaCT++ [8] DIG reasoner. FLC integrates the CONFORMANT-FF conformant planner [7]. PLC uses Castor-based WSDL and BPEL4WS parsers, and exploits the WSYNTH tool [11] to perform datanet-based composition. The remaining code, which includes the various algorithms developed for the architecture as well as the interfacing to the external tools, is realized in Java and ANSI C, for maximum portability. To enable the direct end-to-end utilization of our architecture, we also included a final automated deployment phase for the resulting orchestrator, which assumes the presence of an Active Web Flow engine running over a local Apache Tomcat platform. This setup, together with an ad-hoc embedding of our architecture within a web interface for the reference scenario, provides a convenient and realistic setting to test our tool: the produced orchestrator is given the trip details specified by the user via the web interface, interacts with the (remotely deployed) components that have been discovered by the chain, and return results which are intercepted and presented via the web interface. For our experiments, we deployed remote components implementing non-deterministic behaviors, so that different interaction schemes can be experienced. This way, we empirically checked that the orchestrator responds correctly in all possible cases.

We first tested or architecture in an *orchestration on demand* setting, where an orchestrator is produced "on-the-fly", in response to a specific trip booking request from the user. Notice that since an on-demand orchestrator is tailored to a specific request, it is not required to be fully general. Indeed, in our case, the transportation means required by the user is used to select a specific WSMO requirement which refers to the associated trip subconcept (rather than to the general one), and to generate an orchestrator that handles trips of that kind. The whole execution of the architecture in this setting, using a 700Mhz Pentium 4 machine running Linux, takes about 10 seconds; this is much faster than it would take for a user to look for services, and to interact with each of them. Amongst the various phases, the most time-expensive is PLC, which takes alone about 9 seconds.

We then tested our architecture in a *once for all* setting, where a fully generic orchestrator is built once and for all to deal with any possible user request. This meant running the architecture over the semantic goal stated in Sec.2. Of course, the discovery and composition tasks are harder for a generic orchestrator than for one which is specific to a travel means, since more component services need to be discovered first, and composed then. Indeed, when such a general WSMO goal is considered, the chain takes about 100 seconds to execute, of which 90 taken by PLC; Discovery and FLC, per se, take a negligible time (below one second), while the remaining time is spent for semantic reasoning. This performance is still very good for a once-for-all composition task: to compare, a BPEL4WS programmer would take some hours of work to discover partner services, and (especially) to manually build the BPEL4WS orchestrator.

9 Related and Future Work

We presented and tested an architecture that achieves the long sought after goal of providing fully automated end-to-end composition in a realistic setting, where semantic requirements coexist with imperatively declared service protocols. The architecture combines advanced techniques for Discovery, Functional Level Composition, and Process Level Composition, and therefore significantly departs both from architectures which only focus on the functional description of component services [17, 1], and from approaches based purely on composing processes [9, 3, 15].

Our tests prove the validity of our approach; at the same time, they clearly indicate certain scalability issues, mostly related to the computationally demanding process-level composition phase. In particular, our experiments revealed that the complexity of the datanet requirement plays a very important role, since dataflows turn out to be rather complex networks. Therefore, one possible point of attack to the complexity of the PLC phase stands in improving the datanet requirement extraction procedure, aiming at nets of minimal size. A further direction of work stands in a fuller integration of semantic reasoning within Discovery, to support the extraction of partially matching services in the absence of extensive UDDI-based capability publishing. Finally, we will further investigate our FLC technique, including complexity results for the identified special cases, and broader experiments.

References

[1] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *J. Web Semantics*, 3(4), 2005.

[2] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[3] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*, 2003.

[4] J. De Bruijn, H. Lausen, A. Polleres, and D. Fensel. The Web Service Modeling Language: An Overview. In *Proc. of the 3rd European Semantic Web Conference (ESWC'06)*, 2006.

[5] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[6] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.

[7] Jörg Hoffmann and Ronen Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6–7):507–541, 2006.

[8] Ian Horrocks. The FaCT system. In *Proc. of TABLEAUX-98*, pages 307–312, Berlin, 1998. Springer.

[9] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.

[10] R. Lara, W. Binder, I. Constantinescu, D. Fensel, U. Keller, J. Pan, M. Pistore, A. Polleres, I. Toma, P. Traverso, and M. Zaremba. Knowledge Web research deliverable D2.4.2: Definition of semantics for web service discovery and composition. Technical report, 2004.

[11] A. Marconi, M. Pistore, and P. Traverso. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Proc. of SEFM06*, 2006.

[12] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 2004.

[13] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First Int. Semantic Web Conf.*, 2002.

[14] M. Pistore, L. Spalazzi, and P. Traverso. A Minimalist Approach to Semantic Annotations for Web Processes Compositions. In *Proc. of ESWC'06*, pages 620–634, 2006.

[15] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[16] Evren Sirin and Bijan Parsia. Pellet: An OWL DL reasoner. In *Proc. of the 2004 International Workshop on Description Logics (DL2004)*, 2004.

[17] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.