SWSCE - An Automatic Semantic Web Service Composition Engine

S. Bonomi, V. Colaianni, F. Patrizi, D. Pozzi, R. Russo, M. Mecella
SAPIENZA – Università di Roma, Dipartimento di Informatica e Sistemistica
Via Ariosto 25, 00185 Roma, ITALY
{bonomi,colaianni,patrizi,pozzi,russo,mecella}@dis.uniroma1.it

ABSTRACT

In several scenarios, Semantic Web technologies are gaining momentum as the most promising ones to address the issue of integrating services among different entities, possibly belonging to different location. In particular, Semantic Web Service composition can be used when no individual available service can satisfy a specific client request, but (parts of) available services can be composed and orchestrated in order to do it. In this paper we describe SWSCE, a Semantic Web Service Composition Engine, able to automatically performs the composition of Semantic Web Services.

1. INTRODUCTION

The promise of Web services is to enable the composition of new distributed applications/solutions: when no available service can satisfy a client request, (parts of) available services can be composed and orchestrated in order to do it. Service composition involves two different issues: the synthesis, in order to synthesize, either manually or automatically, a specification of how coordinating the component services to fulfill client requests, and the orchestration, i.e., how executing the previous obtained specification by suitably supervising and monitoring both the control flow and the data flow among the involved services.

Typically, a client continuously interacts with the Web service by asking for something (i.e., the execution of some action returning information to the client), waiting for the reply from the Web service and then asking again until the client reaches its satisfaction. After each step, the client can choose the next operation to invoke among the ones that the service allows at that point of the interaction. All these operations and the constraints on their invocation represent the behavior of the service and are described in the exported service as *semantic description*. In particular, the *behavioral schema* describes all the possible sequences of actions that characterize the service execution, and a *conversation* is a sequence of invocations of operations in a particular order satisfying the constraints imposed by the schema.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

In order to represent the behavioral schema it is possible to use a *transition system* (TS). It is identified by *(i)* the set of actions that the Web service is able to perform, *(ii)* the set of states that the interaction with the service passes through, *(iii)* the initial state and final states of the interaction and *(iv)* the transition relation, which describes state changes as result of action executions. Note that a final state is a state where the client can either ask for another action (moving to another state) or safely stop the interaction with the service. We assume the initial state is always final, so to allow the client to not even start the interaction.

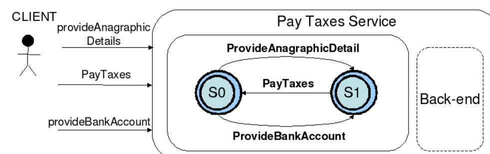

Figure 1: Conversational Model: example of interaction

Figure 1 depicts an example of the conversation model we adopt [2]. A typical interaction is as follows. The Pay Taxes Service is composed by two states S_0, that is an initial and final state, and S_1 that is a final state. The actions that the client can ask for are three: *(i)* `provideAnagraphicDetail` with which the client provides to the service its data and moves from state S_0 to state S_1, *(ii)* `provideBankAccount` with which the client provides to the service its bank account information and moves from state S_0 to state S_1 and *(iii)* `payTaxes` with which the client pays the amount of the tax and he goes back to S_0 from S_1.

In some cases a client would like to interact with a non existing Web Service. Nonetheless, a suitable combination of available services (*component services*) may satisfy the client's needs. We refer to the service obtained as combination of other ones as *target Web service*, and the process of building the target Web service as composition. The interesting challenge is how to make all of this in an automatic way.

Semantic Web technologies, such as OWL/OWL-S [1] and WSMF [2] allow the description of the semantics of entities, including Web services, in a machine-processable format based on XML, and therefore give the right solution

[1] `http://www.w3.org/Submission/OWL-S/`
[2] D. Fensel, C. Bussler: The Web Service Modeling Framework – WSMF. `http://www.swsi.org/resources/wsmf-paper.pdf`

for expressing the behavioral schema of a Web service, by enabling its use in possible compositions. In particular, in WSMF, the behavior of a service is natively represented as a TS and therefore Semantic Web services (SWSs) expressed according to WSMF are well suited to be composed according to the techniques presented in [2, 5].

The contribution of this paper is to present a tool, namely SWSCE – Semantic Web Service Composition Engine, able to perform automatic composition of SWSs. In particular the description of the implementation of the tool is the main focus of the paper. The paper is organized as follows: in Section 2 the semantic composition engine is described; in particular Section 2.1 describes its architecture, Section 2.2 underlines issues related to automatic composition and Section 2.3 shows the tool functionalities through a practical case. Section 3 presents a discussion on the state of the art, and finally Section 4 concludes the paper. At the end Appendix A provides some background information on WSMF, whereas Appendix B shows some code snippets about the running example presented throughout the paper.

2. SEMANTIC WEB SERVICE COMPOSITION ENGINE (SWSCE)

The Semantic Web Service Composition Engine (SWSCE) is a tool to perform automatic composition of SWSs and it can be used during the design phase of new services. In our model, in order to compose existing (available) services, we need to perform the following steps:

- selecting some services from the available ones;
- defining a target service which represents the new service satisfying the user needs;
- executing the composition algorithm and, if successful, create a new SWS.

In its current implementation, the tool works using services described with WSML – Web Service Modeling Language [11]. Some background information on WSML and other elements in WSMF can be found in Appendix A. In this section we will show the architecture of our tool and we will explain its working through a case study.

2.1 SWSCE: Architecture

In Figure 2 the SWSCE architecture is shown. SWSCE consists of three main blocks: *(i)* the *User Interface*, *(ii)* the *Core Engine* and *(iii)* the *Repository Manager*.

The *User Interface* makes available the composition engine to the end user by mean of a user friendly interface. In the current implementation, SWSCE is developed as a WSMO Studio [3] plug-in using the JFace/SWT library. Actually it appears as a button inside the WSMO Studio and it guides the developer in the composition process as a wizard.

The *Core Engine* realizes all the logic needed to perform the computation. This component can be further divided into sub-blocks, namely the Composition Algorithm, the Knowledge Base Manager and the Parser.

The Composition Algorithm. This component is the one responsible to perform the composition. The reasoner that

[3] http://www.wsmostudio.org/

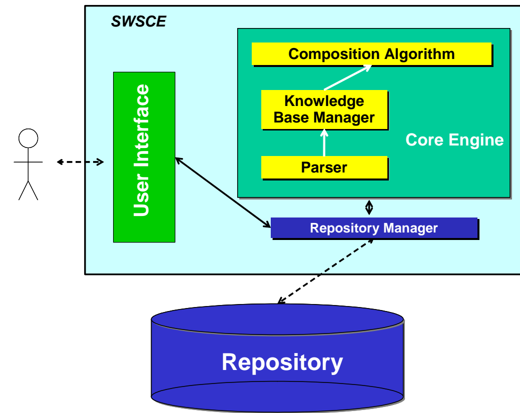

Figure 2: SWSCE Architecture

actually perform the calculation of the composition is TLV. TLV (Temporal Logic Verifier) [21] is a computer aided verification/synthesis system for temporal logic specifications [19, 20]. For a formal description of the simulation-based composition algorithm, we refer to [4, 22], where theoretical bases to address the composition problem, along with some extensions, are built.

The first step is to calculate an object called Community TS. The Community TS is the TS that represents the behavior of the whole community (i.e., set of available services which are candidate for the composition). It can be seen as a global view of the community. Each state of the Community TS is a tuple in which are encoded the states of each available service of the community. Of course the initial state of this TS is represented by all the initial states of the available services. The community TS can execute a transition iff there exists one service of the community that can do that, and so it moves to the next state according to the transition performed by such a service.

This object is exponential in the size of the TSs of the available services, in fact we have to combine all states of the available services. TLV indeed efficiently realizes such cartesian product of the available services.

Then TLV checks the existence of a simulation relation between the Community TS and the target. The simulation relation is a relation on the states of two TSs. By definition two states a,b of two TSs are in simulation relation when *(i)* if a is final, then b is final; *(ii)* if a can make a transition to a', then also b can make such a transition to b', with the same action; *(iii)* a' and b' are still in a simulation relation. Particular relevant is the concept of maximum simulation: the initial state of the target TS is simulated by the initial state of the Community TS. A composition of the services in the community realizes the target service iff the target is simulated by the TS of the community.

Last step of TLV is the extraction of an Orchestrator Generator (OG) from the maximal simulation. Let's have the target service simulated by the Community TS, the orchestrator generator is a TS where: *(i)* the states are tuples including Community TS's as well as target service's state, of course the initial state of the OG is the situation where both target and available services are in their initial state; *(ii)* for each transition of the OG we have the service selec-

tion function that gives the information on which services can execute that transition. Therefore the OG is a TS that, at each point, given an action that the client may request, according to the target service behavior, gives back the set of index of services of the community that can execute the required action.

As an alternative to TLV, a Description Logic Reasoner can be used [1]. In order to generate an orchestration, we need to have a representation of the composition output. The PDL approach [2] consists in writing a big Propositional Dynamic Logic (PDL) formula that represents the required target behavior. In this formula, we encode the behavior of the services, both available ones and target, and the constraints that have to be satisfied in order to guarantee that the service we are going to synthesize will be correct. Moreover the satisfiability of this formula is checked, and in the positive case a TS is built.

Thanks to the correspondence [23] between PDL and Description Logic (DL) we can use a DL Reasoner to obtain the model of the PDL formula, i.e., the composition TS. At the state of the art there are some very efficient tools such as Racer Pro [4], Fact++ [5], and Pellet [6]. Unfortunately, the former two are not useful for our aims, as they can check formulas for satisfiability but return no formula's model, if any, which is the basic structure to extract an actual solution. Pellet provides such a model but it is incomplete: it misses some basic information for extracting a composition. This is due to many optimization techniques that the tool implements. In our experimentation of Pellet, we tried to disable this option in order to make a trade off by having less performance but more information. The result was a big downgrade of performances while information were still missing .

For a proof of concept, we implemented our own DL Reasoner, namely ESC (E-Service Composer) [1]. ESC provides a good model, usable for the composition, indeed it gives the composition of the target as a Mealy TS. Issues with ESC are that: *(i)* it is still a prototype, it lacks all optimization techniques the other tools have, and it isn't very stable; *(ii)* it deals only with deterministic services, that are, services with a deterministic behavior, modeled with a deterministic TS. Indeed ESC checks the satisfiability of DPDL (Deterministic PDL) that is a subset of PDL.

TLV instead supports the composition of non deterministic available services, and provides good performance with stability. In Table 1 we show the results of some tests performed on various reasoners. The second column "*# states*" shows the sum of all the states of the target and the available services' TSs. This is a critical parameter for the composition algorithm; in fact when the number of states grows, we get problems. As shown in Test 2, the model returned by the Pellet lacks some of important features. Increasing the number of states, as shown in Test 3, Pellet crashes saturating all the Java memory heap without finding a solution. ESC instead works good with test 2, but crashes with test 3 saturating the Java memory heap.

The Knowledge Base Manager. This is the component responsible for preparing TLV input and extracts a composition from the Orchestrator Generator. The input is a set of TSs representing service behaviors, written in an XML ad hoc, namely WSTSL (Web Service Transition System Language).

[4]http://www.racer-systems.com/
[5]http://owl.man.ac.uk/factplusplus/
[6]http://pellet.owldl.org/

Test	# states	Pellet	TLV	ESC
1	9	ok	ok	ok
2	15	incomplete	ok	ok
3	20	out of memory	ok	out of memory

Table 1: Confronting the tools

TLV needs that the formulation of the problem is encoded into the SMV language. We encode the TSs of the services (target and available one) into SMV modules mutually interconnected through some shared variables. All modules represent TSs. Synchronization among modules,i.e., the TSs they represent, is realized by using some shared variables that allow to exchange relevant information. In order that everything evolves in a correct way we add some constrains in temporal logic; in particular we define a *failure* assertion in each available module if *(i)* it is requested to one available service to perform a transition that does not exist in that TS, and *(ii)* whenever the target service reaches a final state, available ones are not in their final states. In the main module we define an invariant property *good*, which corresponds exactly to the negation of *failure*. TLV, then, tries to synthesize a strategy (or orchestrator, or controller) for the problem that forces the execution to evolve in a way such that

1. no available system generates a failure;
2. the target service being in a final state implies each available service being in a final state too.

Once the OG is obtained, we easily extract from it one composition, choosing one service index from the set proposed by the OG. The composition is saved into a WSTSL document, that represents the TS of the composition, ready to be parsed for the orchestrator, and to be stored in the orchestration engine (WSMX in our case).

Having an XML language to represent TSs is very useful. If there will be the need of changing the reasoner for our tool in the future all we have to do is just to rewrite the parser for preparing the knowledge base for the new reasoner. Also we decided to use WSTSL to represent the TS of the composition because if there will be the need of changing the execution environment for the orchestration, the tool will be ready by just changing one parser: from WSTSL to the input of the new execution environment. This provides a great extendibility to our tool, making it simple to change an orchestrator, and suitable for possible reasoners available in the future.

The Parser Component. It performs the translation between the WSML representation of the services and the WSTLS representation of the services, and back from the WSTLS TS of the composition to the WSML representation of the orchestration.

The *Repository Manager* (RM) is the block responsible for retrieving and storing the services. Actually the repository used is the one embedded in WSMX and then the RM has to deal with it in order to *(i)* retrieve available services and using part of them, or whole, as input for the composition

and *(ii)* store the orchestration of the composite service if it is found.

2.2 Related Issues

Services composition is not a stand alone process but implies also to search services, to solve semantic incompatibilities, to store results, etc. In this section we discuss two of the main issues related to Semantic Web service composition: *discovery* and *mediation*.

When we want to compose SWSs, we have to select a group of them that will identify the community of the candidates services. To build the community it is necessary to discover all the possible relevant services for the composition, may be simple for a domain expert or for a service expert, but it could be not so easy when we want to automate this process. In the general case, an automatic discovery engine eases the task of reasoning about the service signature (semantics of the I/O parameters, etc.) in order to select the most suitable community. The tool presented in this paper does not use any discovery engine but leaves to the service engineer the task of selecting the services that will act as the input for the composition. Currently we suppose that SWSCE is used by expert users.

Composition of services considers only the conversations of the services and does not take into account the input data format. When services are defined in different ontologies, then there could be the need of a mediation process. The mediation process allows different services to represent the same concepts in spite of possible differences in the ontology representation. Data mediation concerns the transformation of the syntactic format of the messages exchanged by services while the ontology mediation concerns the transformation of terminology used inside the message exchange. Our tool does not solve directly the mediation problem but it inserts inside the synthesized orchestration placeholders where to insert possible mediators. In Section 2.3 we show a concrete example.

2.3 SWSCE: a Practical Case

In this section we presents a case study in order to understand how SWSCE works. A service engineer decides to realize a SWS in which it is possible to pay government taxes online by providing either bank account number or personal data. First of all, he will define which is the behavior of the new SWS, by designing the TS (cfr. Figure 3).

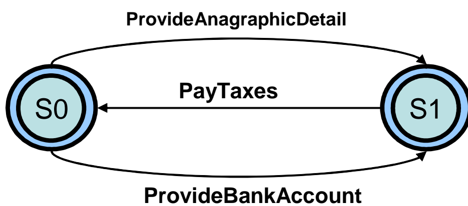

Figure 3: Target Service TS

The engineer writes the choreography of the desired service. In WSMO (cfr. Appendix A) all the messages related to an operation of a SWS are considered as *concepts*. Those concepts are used inside the *choreography* description in order to declare the *state signature* by means of the request and response messages. In the *state signature* all the messages exchanged during the service invocation are described. More precisely, in the *in* statement the concepts which represent the *request* message for the service are declared. While in the *out* statement the concepts which represent the *response* message returned by the service are declared. This signature is the same for all states. The elements that can change and that are used to express different states of a choreography are the instances (and their attribute values) of concepts. Thus, a specific state is described by a set of explicitly defined instances and values of their attributes [24].

From a WSMO perspective, the behavior is described inside the choreography. The conceptual model for WSMO choreography is based on Abstract State Machines (ASMs). A choreography in WSMO inherits the principles of ASMs by defining *(i) a set of states*, where each state is described by an algebra and *(ii) guarded transitions* to express the state change by means of updates. In this first version of SWSCE in order to handle all the requests coming from a client and to perform the update operations, the transition condition using the *forall* statement have to be declared: **forall** *Variable* **with** *Condition* **do** *Rules(Variable)* **endForall**. The meaning of such a rule is to execute simultaneously the enclosed *Rules* for each variable in *Variables* satisfying the *Condition* of the rule where typically a variable will have some free occurrences in the rules which are bound by the quantifier [24].

For a better explication of the sequence of actions performed by the service during an invocation, consider the choreography of the *payTaxByBankAccount* service shown in Appendix B.

Now the service engineer can start using SWSCE by clicking the SWSCE button of the user interface (Figure 4).

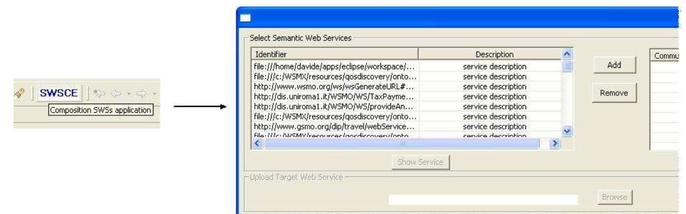

Figure 4: Starting SWSCE

SWSCE starts by retrieving the SWSs through the RM from the repository and then shows their identifiers in the Community table (Figure 6). It can also possible to show the WSML of the SWSs by clicking on the button `Show service`. The Community table contains the list of the component SWSs participating in the composition, i.e., in our example the *payTaxesProvideBankAccountWS* and *payTaxesProvideAnagraphicalDetailsWS* whose behavior is represented in Figure 5.

The SWS previously designed is now selected (Figure 7) as the target service that the user wants to obtain as result of the composition.

When the `Start` button is clicked (Figure 9), SWSCE starts to elaborate the choreographies of the component SWSs by parsing them. If such a composition can be obtained a new SWS is created. The new composed service obtained has the choreography of the target service and it is followed by a *skeleton* of a possible orchestration. The Figure 8 shows the structure of the obtained orchestration.

In every transition rule an intermediate state has been

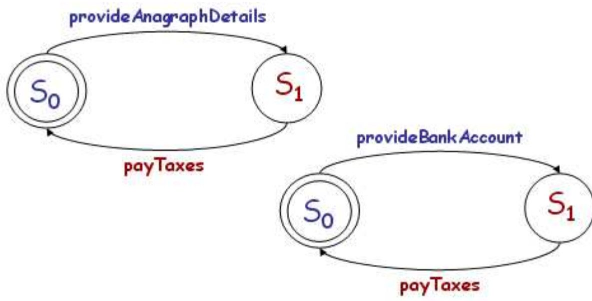

Figure 5: Operations of component services

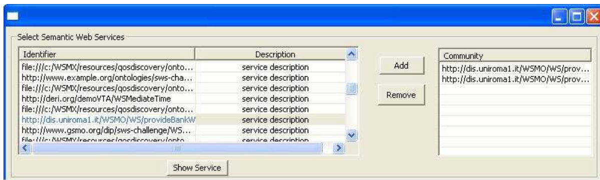

Figure 6: Select candidate Services

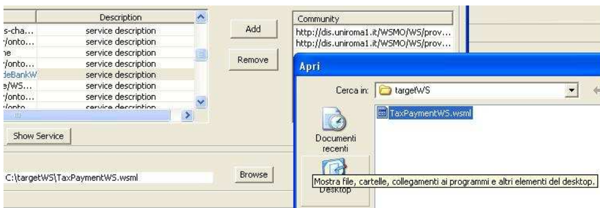

Figure 7: Select Target Service

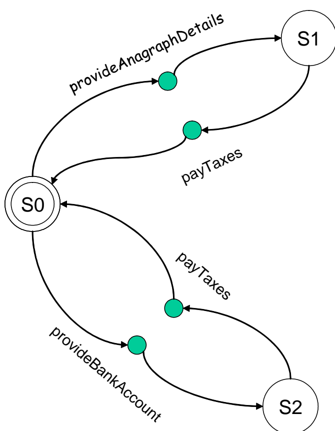

Figure 8: Orchestration with intermediate states

defined by default from the algorithm. In this state possible mediators have to be declared, which allow the transition from a source state to the target state. In the obtained WSML file, the intermediate state contains a "placeholder" in which it is possible to declare the mediation:

```
abstractStateMachine transitionRules
  forall {?cs} with (?cs[oasm#value hasValue oasm#S0]
  memberOf oasm#controlState) do
     //implement "perform" to invoke service PayTaxesByBankAccount
     delete (?cs[oasm#value hasValue oasm#S0])
     add (?cs[oasm#value hasValue S0_mediation_S2])
 endForall

 forall {?cs} with (?cs[oasm#value hasValue oasm#S0_mediation_S2]
 memberOf oasm#controlState) do


     // implement (if needed) "mediation" for
     // provideBankAccountRequest/provideBankAccountResponse


     delete (?cs[oasm#value hasValue oasm#S2_mediation_S0])
     add (?cs[oasm#value hasValue S2])
     endForall
```

The WSML file of the composed service has the choreography of the target service and the orchestration obtained by the composition. Furthermore it is possible to store the composed SWS obtained by means of the composition through the RM inside the repository clicking on the button `Store Service` (Figure 10).

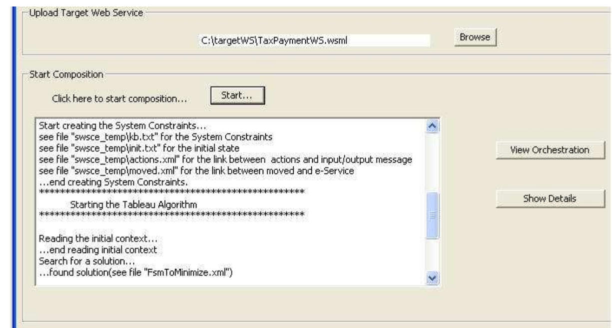

Figure 9: Starting the Composition

3. DISCUSSION

The use of Semantic Web service technologies is gaining a special role in building complex applications, e.g. as in the context of eGovernment services. In order to make it possible, many efforts in the scientific community are going in the direction of automatic services composition in order to build more complex services using the simpler building blocks as component services. As we have shown in our work, in order to add semantic description to the services (in particular for their behaviors), we need to leverage an ontology description. In the eGovernment domain, this approach is growing in importance and [25] provides an ontology (GEA ontology) that models the Public Administration domain using WSMO.

In order to discuss automatic services composition, and to compare different approaches, [3] introduces a conceptual framework for "semantic service integration", consisting of:

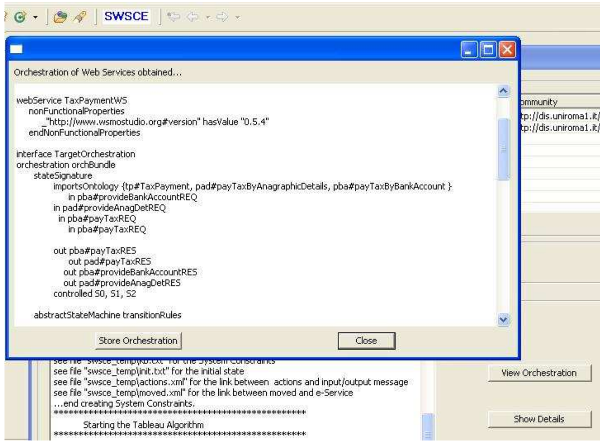

Figure 10: Storing the Orchestration of the Composite Service

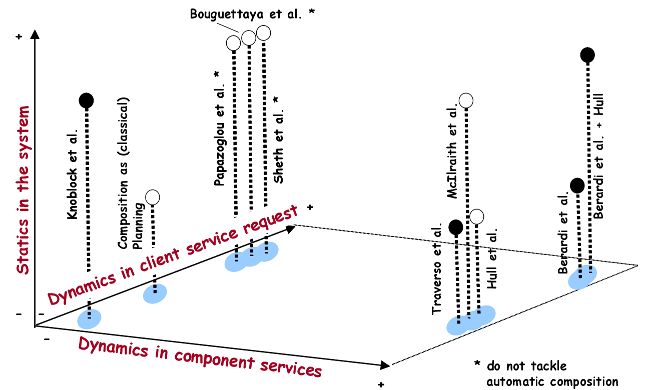

Figure 11: Comparison of the various approaches to automatic service composition

- the *community ontology*, which represents the common understanding on an agreed upon reference semantics between the services concerning the meaning of the offered operations, the semantics of the data flow through the service operations, etc;
- the *set of available services*, which are the actual Web services available to the community;
- the *mapping* for the available services to the community ontology, which expresses how services expose their behavior in terms of the community ontology;
- and the *client service request*, to be expressed using the community ontology.

Generally, the community ontology involves several aspects: on one side, it describes the semantics of the information managed by services, through appropriate semantic standards and languages (e.g., WSMO); and on the other side, it also consider some specification of the service behaviors, on possible constraints and dependencies between different service operations, not limited solely to pre- and post-conditions, but also considering the process of the service.

In building such a "semantic service integration" system, two general approaches can be followed.

- In the *Service-tailored* approach, the community ontology is built mainly taking into account the available services, by suitably reconciling them. Indeed the available services are directly mapped as elements of the community ontology, and the service request is composed by directly applying the mappings for accessing concrete computations.
- Conversely in the *Client-tailored* one, the community ontology is built mainly taking into account the client, independently from the services available.They are described (i.e., mapped) by using the community ontology, and the service request is composed by reversing these mappings for accessing concrete computations.

Much of the research on automatic service composition has adopted, up to now, a service-tailored approach, examples are the works based on Classical Planning in AI (e.g., [26],[6]), the works of Papazoglou's et al. [27], Bouguettaya at al. [16], Sheth et al. [9, 8], the work of McIlraith at al. [15] and the work by Hull et al. [7, 14].

A very little research has been done following a client-tailored approach, but some remarkable exceptions should be mentioned: the work of Knoblock at al. [17] the work of Traverso et al. [18], the work of Hull et al [12] and finally [2, 5].

Figure 11 summarizes the considered works (more details in [3]). The three axis represent the levels of detail according to which the community ontology and the mappings and the client request can be modeled. Namely, *(i)* statics in the system represents how fine grained is the modeling of the static semantics (i.e., ontologies of data and/or services, inputs and outputs, alphabet of actions, etc.); *(ii)* dynamics in component services represents how fine grained is the modeling of the processes and behavioral features of the services (only atomic actions, transition systems, etc.); and *(iii)* dynamics in client service request represents how fine grained is the modeling of the process required by the client, varying from a single step (as in the case of services consisting essentially in a single data query over a data integration system) to a (set of) sequential steps, to a (set of) conditional steps, to including loops, and up to running under the full control of the client. Black/white lollipops represent service-based (white) vs. client-based (black) approaches.

The approach proposed in this paper is the first result, with respect to the client-tailored approach, which uses WSMO to describe Semantic Web services and that proposes a tool for automatic service composition, by providing it as a WSMO/WSMX plug-in.

4. CONCLUSION AND FUTURE WORK

In this paper we have presented SWSCE, a tool for automatic composition of Semantic Web services and we have shown how our tool works like a wizard. Our tool actually works by retrieving SWSs through the RM in the repository embedded inside WSMX. Currently we are also developing a new version working with a different distributed discovery engine. Actually, our tool provides support for the mediation by returning an orchestration where it is possible to add manually the mediation, but in the next version we are trying to add directly in the wizard the possibility to start

the mediation tool and to define directly the mapping at the end of the composition process.

Acknowledgements. The work presented in this paper is partly supported by the IST FP6 project *SemanticGOV*. The authors would like to thank the project partners for their useful comments and discussions on SWSCE.

5. REFERENCES

[1] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. ESC: A Tool for Automatic Composition of Services Based on Logics of Programs. In *Proc. VLDB-TES 2004*, pages 80–94.

[2] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Service Composition Based on Behavioral Descriptions. *International Journal on Cooperative Information Systems*, 14:333–376, 2005.

[3] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Automatic Web Service Composition: Service-tailored vs. Client-tailored Approaches. In *Proc. AISC 2006 (jointly with ECAI 2006)*.

[4] D. Berardi, F. Cheikh, G. De Giacomo, and F. Patrizi. Automatic Service Composition via Simulation. *International Journal of Foundations of Computer Science (IJFCS)*, 19:429–451, 2008.

[5] D. Berardi, G. D. Giacomo, M. Mecella, and D. Calvanese. Composing Web Services with Nondeterministic Behavior. In *Proc. ICWS 2006*.

[6] J. Blythe and J. Ambite, editors. *Proc. ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[7] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: a New Approach to Design and Analysis of e-Service Composition. In *Proc. WWW 2003*.

[8] J. Cardoso and A. Sheth. Introduction to Semantic Web Services and Web Process Composition. In *Proc. 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*.

[9] F. Curbera, A. Sheth, and K. Verma. Services Oriented Architectures and Semantic Web Processes. In *Proc. ICWS 2004*.

[10] J. de Bruijn, D. Fensel, U. Keller, and R. Lara. Using the Web Service Modelling Ontology to Enable Semantic eBusiness. *Communications of the ACM*, 2005.

[11] J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel. The Web Service Modeling Language WSML: An Overview. In *Proc. ESWC 2006*.

[12] C. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated Composition of E-Services: Lookaheads. In *Proc. ICSOC 2004*.

[13] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX - A Semantic Service-Oriented Architecture. In *Proc. ICWS 2005*.

[14] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a Look Behind the Curtain. In *Proc. PODS 2003*.

[15] S. McIlraith and T. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. KR 2002*.

[16] B. Medjahed, A. Bouguettaya, and A. Elmagarmid. Composing Web Services on the Semantic Web. *Very Large Data Base Journal*, 12(4):333–351, 2003.

[17] M. Michalowski, J. Ambite, S. Thakkar, R. Tuchinda, C. Knoblock, and S. Minton. Retrieving and Semantically Integrating Heterogeneous Data from the Web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.

[18] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI 2005*.

[19] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive Designs. In *Proc. VMCAI 2006*.

[20] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. POPL 1989*.

[21] A. Pnueli and E. Shahar. The TLV System and its Applications, 1996.

[22] S. Sardiña, G. De Giacomo, and F. Patrizi. Behavior Composition in the Presence of Failure. In *Proc. KR 2008*.

[23] K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. IJCAI 1991*.

[24] J. Scicluna, A. Polleres, D. Roman, and D. Fensel. Ontology-based Choreography and Orchestration of WSMO Services. Technical report, Digital Enterprise Research Institute (DERI), 2006.

[25] X. Wang, T. Vitvar, V. Peristeras, A. Mocan, S. K. Goudos, and K. A. Tarabanis. WSMO-PA: Formal Specification of Public Administration Service Model on Semantic Web Service Ontology. In *Proc. HICSS-40*, 2007.

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

[27] J. Yang and M. Papazoglou. Service Components for Managing the Life-cycle of Service Compositions. *Information Systems*, 29(2):97–125, 2004.

Appendix A - Introduction to WSMO

In our approach, the implementation of the Semantic Web Service composition is realized by means of the Web Service Modelling Framework (WSMF) that provides an ontology (WSMO) [10] for the definition of the domain of the services provided and of the elements characterizing each SWS, that is *(i)* ontologies, *(ii)* goals, *(iii)* services and *(iv)* mediators; a language (WSML) [11] for the formal description of the syntax and the semantics of all the elements defined in WSMO; and an execution environment (WSMX) [13] for the deploy, execution, composition and orchestration of the Semantic Web services.

Even if in the present paper we will focus mainly on the description of the behavior of the SWSs, it is important to briefly introduce the definition of the basic elements of a SWS in the WSMO ontology.

A SWS description in WSMO is defined in terms of:

- the *concept* representing the basic elements of the agreed terminology concerning the domain the SWS belongs to. A concept represents classes of objects of a real or abstract world that have a specific shared property (e.g., being a tax to be paid) and well defined attributes. An example of the semantic of the `pay-`

`TaxesByAnagraphicDetails` service is described in the WSML code shown below.

```
importsOntology
       _"http://www.semantic-gov.org/Italy#ItalianCitizen"

concept AnagraphicDetails
        name impliesType  (1 1) _string
        dateOfBirth impliesType  (1 1) { _string, _dateTime}
        SSN impliesType  (1 1) _string
        surname impliesType  (1 1) _string

concept Tax
        payToOrganization ofType _string
        amount ofType _string
        payBy ofType _string
```

- the *interface* describing the behavior of the SWS; in particular the interface consists of two kinds of information: the *choereography* describing how a client (be it a human or a software) can interact with the service in order to use its functionalities, and the *orchestration* describing the interactions among the given service and other SWSs in order to achieve the target functionalities.

  In Figure 12 the different WSMO Choreography and Orchestration perspectives are illustrated.

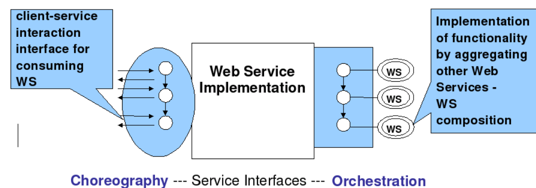

Figure 12: WSMO choreography and orchestration perspectives.

The choreography describes the behavior of the service, that is, how client communicates with the service in order to consume the provided functionalities. WSMO choreography is based on the Abstract State Machines (ASM) formalisms (providing the basis for the description of both the choreography and orchestration); an ASM consists of two components: the state and the guarded transitions. The states are represented by an ontology whose content is dynamic while guarded transitions based on the model **if** *Cond* **then** *Update* specify transitions between states (that is changes of states).

The orchestration consists in the description of how the overall functionality of the SWS is achieved through the cooperation of different service providers. In WSMO, orchestration is based on the same ASM formalism but the guarded transitions are extended to support the use of *mediators*, that is software components providing data links among different services.

Appendix B

In the following there is the choreography of the *payTaxByBankAccount* service:

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
namespace { _"http://dis.uniroma1.it/WSMO/WS/provideBankWS#",
     pba _"http://dis.uniroma1.it/WSMO/payTaxesByBankAccount#",
     ctrfsm _"http://dis.uniroma1.it/WSMO/ControlFsm#",
     wsmostudio _"http://www.wsmostudio.org#" }

//..
interface byBankInterface
choreography _#
stateSignature byBankStSign
     in
          concept pba#provideBankAccountRE
                   concept pba#payTaxREQ
     out
          concept pba#provideBankAccountRES
                   concept pba#payTaxRES
transitionRules _#
     forall {?controlstate, ?provideBankAcc} with (
                 ?controlstate[ctrfsm#currentState hasValue 0]
                 memberOf ctrfsm#controlledState and
                 ?provideBankAcc memberOf pba#provideBankAccountREQ
           ) do
                 add(?controlstate[ hasValue 1])
                 add(_# memberOf pba#provideBankAccountRES)
     endForall

     forall {?controlstate, ?payTax} with (
                 ?controlstate[ctrfsm#currentState hasValue 1]
                 memberOf ctrfsm#controlledState and
                 ?payTax memberOf pba#payTaxREQ
           ) do
                 add(?controlstate[ hasValue 0])
                 add(_# memberOf pba#payTaxRES)
     endForall
```