Development of Semantic Web Services at the Knowledge Level

[1] Departamento de Inteligencia Artificial, Facultad de Informática.
Campus de Montegancedo s/n, Universidad Politécnica de Madrid.
28660 Boadilla del Monte, Madrid. Spain.
asun@fi.upm.es, rgonza@delicias.dia.fi.upm.es

[2] Departamento de Electrónica e Computación, Facultad de Física.
Campus Universitario Sur s/n, Universidade de Santiago de Compostela.
15782 Santiago de Compostela, A Coruña.
lama@dec.usc.es

Abstract. Web Services are interfaces to a collection of operations that are network-accessible through standardized XML messaging, and whose features are described using standard XML-based languages. Semantic Web Services (SWS) describe semantically the internal structure and the functional/non-functional capabilities of the services, facilitating the design and evaluation of SWSs based on that semantic description of the features of the services. To enable users to design and compose SWSs at the knowledge level, the ODE SWS framework has been proposed. That framework uses problem-solving methods to describe the functional and structural features of the SWSs. In this work, we present a description of the ODE SWS environment as an implementation of the ODE SWS framework. Specially, we focus on the description of the capabilities of the SWSDesigner, the tool of the ODE SWS environment that enables users to design graphically SWSs through different but complementary views of the services.

1 Introduction

Web Services (WSs) are interfaces that describe a collection of operations that are network-accessible through standardized Web protocols, and whose features are described using a standard XML-based language ([1,2]). These features are the following: (1) communication features describe the protocols required to invoke the service execution; (2) descriptive features detail the e-commerce properties; (3) functional features that specify the capabilities, enabling thus for an external invoking agent to determine whether the service execution can obtain the requested results; and (4) structural features describing the internal structure of a composite service, that is, which are its structural components and how those components are combined among them to execute the service.

In this context, the Semantic Web [3] has risen as a Web evolution where the information would be directly machine-readable to enable software agents to access to it. Following this approach, Web Services in the Semantic Web, so-called Semantic

L.-J. Zhang and M. Jeckle (Eds.): ECOWS 2004, LNCS 3250, pp. 72–86, 2004.

Web Services (SWSs), must be described using an ontology that is expressed in a semantically enriched markup language [4]. This semantic description will facilitate external agents to understand both the functionality and the internal structure of the services to be able to discover, compose, and invoke SWSs [5]. The markup language could be OWL [6], but it must be combined with WS standard languages to be able to use the current infrastructure of the WS [7]. Following this approach, the OWL-S [8] specification (formerly DAML-S [9]) has been proposed to describe services in a semantic manner, using OWL in combination with WSDL [10] and SOAP [11].

However, as a previous step to the specification of SWS in a semantic Web-oriented language, the SWS should be designed at a knowledge or conceptual level [12] to avoid inconsistencies or errors among the services that constitute the SWS. In this context, SWS design consists in specifying the descriptive, functional, and structural features of a service. Currently, there are some proposals to edit/design SWSs, but the main drawback of these available editing tools is that they operate at the representation level, so the following problems may arise:

- These tools are language-dependent, like the WSMO Editor [13], this means that: (1) SWSs designed with these tools are less reusable, because the design can be constrained with the chosen language characteristics, meaning that the basic separation between design-implementation phases is broken; and (2) the designs are more prone to inconsistencies or errors that designs at the knowledge level.
- Many tools that claim to be SWSs editors are not actually more than mere ontology editors, like the widespread option of OWL-S development with the OWL plug-in for Protegé-2000 [14]. This option not only suffers from all the problems enumerated above, but also adds the problem of working with an ontology instantiation, not with a SWS-like structure.

To solve those drawbacks, we have proposed a framework, called ODE SWS [15], for design of SWSs at knowledge and independent-language level. This framework is based on: (1) a stack of ontologies that describe explicitly the different features of a SWS; (2) a set of axioms used to check the consistency and correctness of the ontology instances, and, thus, of the service represented by the ontologies; and (3) the assumption that a SWS is modeled as a problem-solving method (PSM) that describes how the service is decomposed into its components, and which is the control of the reasoning process to execute the service.

In this paper, we describe the architecture of ODE SWS for design of SWSs, and specially, we focus on the capabilities of SWSDesigner, a tool that is the user interface of ODE SWS. SWSDesigner uses PSM-alike views for describing the SWSs and, the service is designed thus by filling the SWS definition; specifying its associated task (interaction and logic diagrams); creating the overall task hierarchy (decomposition diagram); building both the associated method internal dataflow (knowledge flow diagram) and the coordination of the execution of its subtasks (control flow diagram).

This paper is structured as follows: in the following section the ODE SWS framework is presented; then, we show the functionalities of the environment that implements the ODE SWS framework: its architecture and current modules are described, paying special attention to the SWSDesigner graphical editor. Finally, we summarize the main contributions of the paper.

2 ODE SWS Framework

The aim of designing SWSs is making explicit the features previously mentioned and specially each one of its structural components, guaranteeing the correctness of the proposed design, and avoiding the inconsistencies. For that, we will need to perform inferences about the service features to determine whether the proposed design is correct. This means that the service features (and the service itself) should be explicitly and semantically described, and, for it, the use of ontologies seems to be the most appropriate solution. This approach has been also followed by other authors [8], who use a semantic-enriched markup language to create an ontology (OWL-S) that describes the service features.

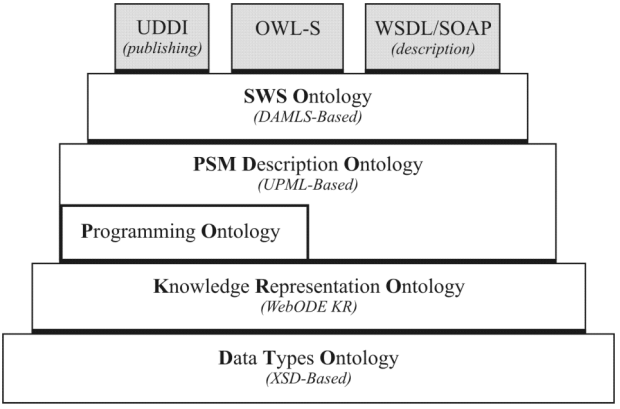

Fig. 1. Ontology set identified in the ODE SWS framework [15] to SWS design. These ontologies have been developed based on well-known specifications and *de facto* standards

Figure 1 shows the stack of ontologies proposed in the ODE SWS framework [15,23] to describe all the features of a SWS (and the service itself). To construct these ontologies well-known specifications or *de facto* standards have been used. This will favor the interoperability of the ODE SWS framework with applications or solutions constructed following one of those specifications.

Problem-Solving Method Ontology

Problem-Solving Methods (PSM) [16,17] are knowledge components reusable among different, but related, domains and tasks. The Unified Problem-solving Method Language (UPML) [18] is a *de facto* standard that describes the components of a PSM as: (1) *tasks*, they describe the operation to be solved in the execution of a method that solves such task, specifying the input/output parameters and the pre/post-conditions (competence) required to be applicable (this description is independent of the method used for solving the task); (2) *methods*, elements that detail the control of the reasoning process to achieve a task; and (3) *adapters* [19] specify mappings among the

knowledge components of a PSM. The adapters are used to achieve the reusability at the knowledge level, since they bridge the gap between the general description of a PSM and the particular domain where it is applied.

Based on the UPML specification we have created a PSM ontology that enhances the description of the UPML elements. Some additions or differences are: new relationships between tasks and methods are defined, a set of program elements to specify the control flow of a composite method are defined, we define a combination of them that allow us to derive several basic workflow-like patterns [20,21], like sequence or exclusive and multiple choice, the domain ontology is managed in a different manner.

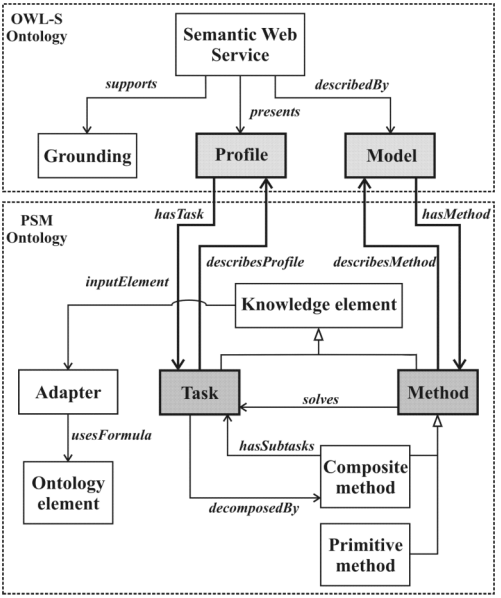

Fig. 2. Semantic Web Service ontology and its relationship with the PSM description ontology, where tasks will be used to represent the functional features of a SWS, and methods to describe the internal structure and control flow of that service [15]

Semantic Web Service Ontology

As Figure 2 shows, SWS ontology replicates the upper-level concepts of the OWL-S ontology, so we consider: (1) *profile* contains all the descriptive and functional features, and it is establishes a relationship with a task of the PSM ontology; (2) *model* deals with the internal structure of the service, which will be executed by a method that defines an operational description to solve the task related to the functional features of the service; and (3) *grounding*, which specifies the access protocol and the necessary message exchanges to invoke the service.

2.1 Framework for SWS Design and Composition

The proposed framework for SWS [15] design and composition is directly based on the stack of ontologies that describe the features of a SWS. The ODE SWS framework details how to create a SWS with the capabilities required by an external agent or user. The main elements of the framework are (Figure 3):

1. *Instance model*. Design of SWSs means to instantiate all the ontologies that describe what a service is: the domain ontology used by the service is instantiated in both data types and knowledge representation ontology, whereas the service features are instances of both PSM and SWS ontologies. The whole instances constitute a model that specifies the SWS at the knowledge level.
2. *Checking model*. Once the instance model has been created, it is necessary to guarantee that the ontology instances do not present inconsistencies among them. Design rules will be needed to check this, particularly when ontology instances have been created automatically.
3. *Translate model*. Although a service is modeled at the knowledge level, it must be specified in a SWS-oriented language to enable programs and external agents to access to its capabilities.

This framework enables the (semi) automatic composition of SWSs using (1) PSM refiners and bridges to adapt the PSM ontology instances to the required capabilities of the new service; and (2) design rules to reject both PSM and SWS ontology instances that present errors or inconsistencies among them. Design rules are used to reduce the service candidates combined to obtain the new service.

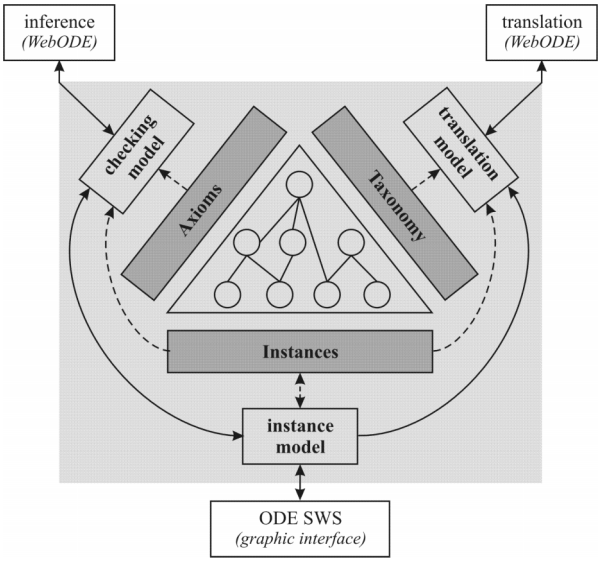

Fig. 3. Framework for design and composition of SWSs based on the ontologies that describe the service features

3 ODE SWS Environment

Following the ODE SWS framework, a highly modularized, scalable and dynamic environment for development of SWSs called ODE SWS [22,23] has been implemented. Its architecture and functionalities will be thoroughly explained in the forthcoming sections.

3.1 ODE SWS Architecture

The architecture of the ODE SWS, as Figure 4 shows, is composed of three main layers, which reflect the layers introduced in a software design [24]:

Presentation Layer. This layer is about how to handle the interaction between the user and the software system. This layer is entirely composed by the SWSDesigner graphical editor that manages a graphical model (SWSGM) of the SWS. Its main functionalities are: (1) the appropriate management and representation of the model; (2) graphical processing of all the possible interactions among the elements that compose such model; and (3) the management of the graphical elements of the ODE SWS environment. Any other functionality is delegated to the appropriate ODE SWS modules of the domain layer.

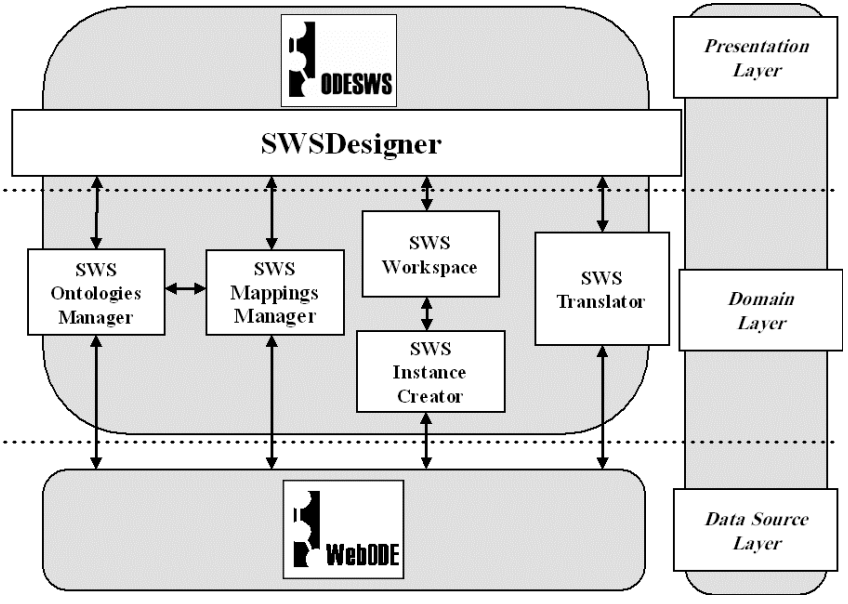

Fig. 4. Software architecture of the ODE SWS environment for the development of Semantic Web Services

Domain Layer. This layer contains all the components that work in the domain of the ODE SWS application (i.e., SWSs), and, consequently, most of ODE SWS modules will be located in this layer. SWSDesigner directly will invoke the execution of the ODE SWS modules to support the execution of an operation needed to guarantee a correct design of the service. The intended functionalities of each of these components are:

- *SWSOntologiesManager.* The purpose of this module is both to offer a uniform manner for accessing to ontologies implemented in different languages, and to enable ODE SWS to access to different repositories of ontologies. Therefore, this module guarantees the language and technology independence for the ontologies managed in the development of the service. Currently this module can manage either ontologies implemented in RDF(S), DAML+OIL and OWL, or ontologies stored in the WebODE platform [25]. Note that this module is part of the domain layer because the ontologies that it manages are stored in the source repository.
- *SWSMappingsManager.* The objective of this module is to manage mappings, which will be (semi) automatically defined between the elements of the task ontologies and the elements of the method and domain ontologies.
- *SWSWorkspace*. While the user is developing a SWS, an incomplete (even inconsistent) service may be stored and managed. This module performs these activities, enabling thus the store and recovery of ongoing SWSs.
- *SWSInstanceCreator*. This module creates the instances of the stack of ontologies that describe the SWSs from the graphical representation of the service generated by the SWSDesigner.
- *SWSTranslator.* This module implements the translation model of the framework. Once the SWS has been modeled at the knowledge level, it must be translated to a SWS-oriented language to enable other programs or agents to understand its capabilities. So, once the SWS has been created using SWSDesigner, the user asks for a list of available translators. After that, this module receives which is the selected translator, the desired output format, and the graphical representation model of the SWS to be translated. SWSTranslator generates the translation and, additionally, it incorporates information about the different problems that may have arisen during the translation process. Note that it may use its own inner translators or even external translation services, as it does with the export services of the WebODE platform. At present, the translation to OWL-S and WSDL are available.

Data Source Layer. In this layer, other applications act on behalf of the ODE SWS environment to provide support for operations do not implemented in the environment. For the ODE SWS environment, this layer will be just composed by the WebODE platform, which will provide services for the management and access to the ontologies used in the development of the SWSs.

3.2 SWSDesigner

SWSDesigner is a graphical editor based on the assumption that the design and development of a service should be performed from different, but complementary, points of view. In the ODE SWS framework, as we have already stated, a service is described instantiating a set of ontologies that describe all the SWS features. Taking this into

account, SWSDesigner provides a user-friendly graphical interface with which the user is completely unaware of the instantiation of the SWS ontologies. In this way, we achieve the following objectives: (1) the service is defined at a high level of abstraction using PSMs to model the service features. This modelling enhances the quality of the design, eases its evaluation and validation processes, and favours its reuse; and (2) this design process is far more simple and less error prone than manipulating directly instances of the different description ontologies.

As stated in the ODE SWS framework section, a service has its own descriptive (non-functional) properties that are described by a task, whereas the description of how this task (and the service) is supposed to be carried out is specified by a method. As we will see, all this information can be gathered from the different views that the SWSDesigner manages. To illustrate the functionalities of this tool, in the following sections we will define a service for selling movie tickets.

Service Definition Panel. The service definition panel includes several kinds of information related to the declarative (or non-functional) service features and the properties of the providers. It includes (see Figure 5):

- *Service definitions*. The information of the service contained in its definition will include: (1) information needed for the identification and description of the service (i.e., name, description and URL); (2) descriptive features that will detail its e-commerce properties (i.e., geographical location, commerce classification, service provider, etc.); and (3) a summary table of services that contains information about all the services in a condensed view.
- *Provider definitions*. The provider forms include information such as: (1) information necessary for the identification and description of the provider (i.e. name, description); (2) descriptive features that detail its e-commerce properties (its geographical radius and code, and its commerce classification); (3) the contact persons that can be assigned (and the ones that have been already assigned) to the providers, including all the contact data (name, phone, e-mail, fax, etc.); and (4) summary table of providers that contains information about all the providers in a compacted manner.

Furthermore, in the service definition panel, the description of the functionality (input/output data) and the competence of the task associated with the service is introduced. This description is achieved with two diagrams:

- *Interaction diagram*. In this diagram the input and output roles of the task are defined. A role is an ontology element (concept or attribute), and it can be easily inserted into the diagram by just dragging-and-dropping them from the ontology tree. Once it has been deployed, an edge between the role and the task can be created; depending on the direction of this connection, the role is an input or an output of the task. Unassigned roles are allowed.
- *Logic diagram*. In this diagram, the pre/post-conditions (competence) required by a task to be applicable, and the effects of the execution of such task in the state of the world are set. All these logical conditions are represented as different kinds of cells of the graph, with edges going from the condition toward the task in the case of pre-conditions, and with connections from the task to the cells when effects and post-conditions are introduced.

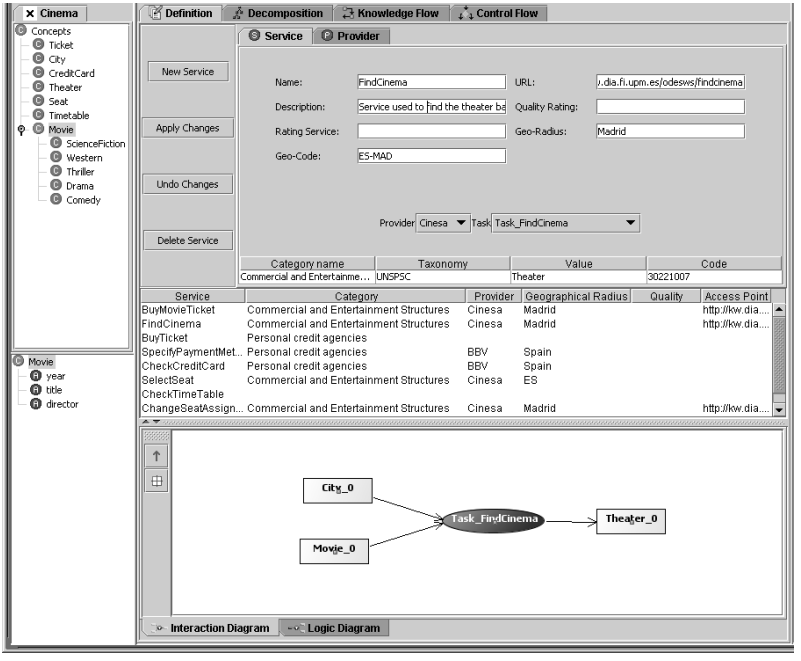

Fig. 5. Service definition panel with the descriptive and functional features of *FindCinema*

In the Figure 5, the service definition panel of the SWS *FindCinema* is shown. All the descriptive information of the service can be introduced here. The task information is introduced in the interaction diagram, indicating that the *Task_FindCinema* task receives two input roles: *Movie_0*, which is an instance of the concept *Movie*, and *City_0*, which is an instance of the concept *City*. The output is an instance of the concept *Theater*. These elements have been dragged-and-dropped from the ontology trees; one of them shows the concepts, and the other shows the attributes of each concept. Note that this ontology could be other ontology different of the domain ontology. As it will be explained, mappings between domain ontology elements and the inputs/outputs of the task are set in the knowledge flow diagram.

Decomposition Diagram. The decomposition diagram enables the user to specify the task hierarchy, that is, how the defined tasks are related to each other. This diagram will be synchronized with the task tree, which will be visible for every SWSDesigner view in the top right corner of the screen. There will be two possible representation options:

– *A task can be decomposed by a method in one or more subtasks*. This view allows user to specify the subtasks in which a task is decomposed by a composite method that solves such task. This kind of relations is represented as an edge from the subtasks to the task.
– *A task specialises another task*. In this case, the user will have to introduce an adapter, more precisely a task-refiner, among the implied tasks. The properties of the task-refiner could be edited by the user after its inclusion in the diagram.

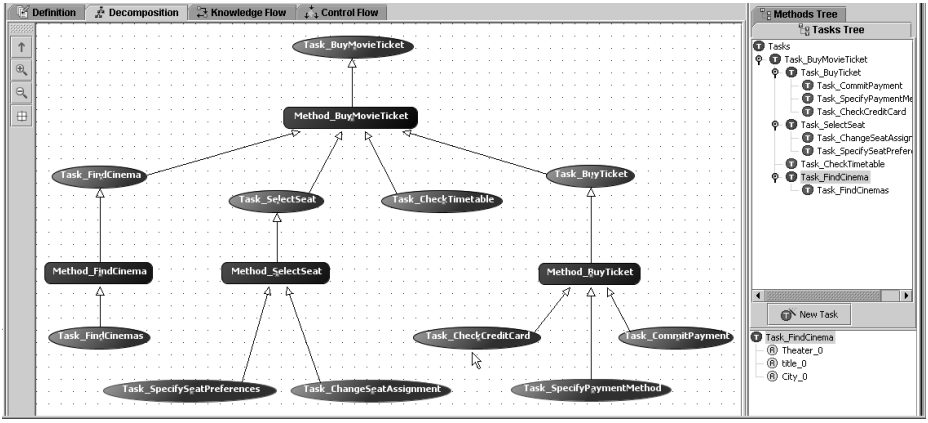

Fig. 6. Decomposition diagram for the *Task_BuyMovieTicket* task that describes the functional features of a given SWS

Figure 6 shows the overall task hierarchy for the selling movie ticket example. Thus, to solve the composite task *Task_BuyTicket*, the tasks *Task_CheckCreditCard*, *Task_CommitPayment*, and *Task_SpecifyPaymentMethod* might need to be accomplished. In the right corner of the screen, the task tree is shown. It allows the access to the tasks in the other views. Note that this view does not impose any order of execution at all; it is set in the control flow diagram.

Knowledge Flow Diagram. The knowledge flow diagram shows the data flow among the tasks in which a given composite method has been decomposed. It relates (1) the inputs/outputs of a method to the inputs/outputs of its various component subtasks; (2) the inputs/outputs of a subtasks of a given method with the inputs/outputs of the other subtasks of such method; and (3) the domain ontology to the method and tasks. Therefore, there will be three basic elements: roles, tasks and edges. Tasks can be placed easily, just dragging-and-dropping them into the graph from the tasks tree. The management of roles and edges must be made more carefully. Roles will wrap an ontology element, and can be inserted into the diagram by just dragging-and-dropping them from the ontology tree. Depending on different possible combinations, three plausible scenarios are distinguished:

- *Unassigned Roles*. When a role is isolated, that is, there are no connections either going out from it or pointing at it, the role does not affect the model at all. This kind of roles is allowed just for user convenience.
- *Inner Roles*. This situation occurs when a role is input of a task and at least output of another task. It might be viewed as an inner message between elements inside the method. Here, a domain-task bridge, which maps the ontology element wrapped by the role with the input/output of the affected tasks, is created.
- *External Roles*. These roles are either inputs or outputs of a task, but not inputs of a task and outputs of another tasks. Therefore, these roles must be provided to the method, if it is going to be invoked, because it is not a data flow between tasks, but these roles are related to output and output messages from or towards the method.

Two adapters are created in this situation thus, a domain-task bridge (as in the case of the inner role) and a domain-method bridge.

To ease the management of the different kind of roles and the different mappings that may appear, the knowledge flow diagram offers two useful tools:

– *Summary diagram*. This diagram is automatically updated each time a user does some kind of manipulation in the knowledge flow diagram (a role is inserted, an edge is removed, etc.). It shows just the external roles, allowing the special and separate handling of them.
– *Mapping generator*. When a user creates an edge from a role to a task (or *vice versa*), an adapter between this role and a role of the task interaction diagram is created. The role of the interaction diagram origin or destination of this binary relation must be directly asked via interface to the user. The mapping generator will obtain automatically the plausible candidates.

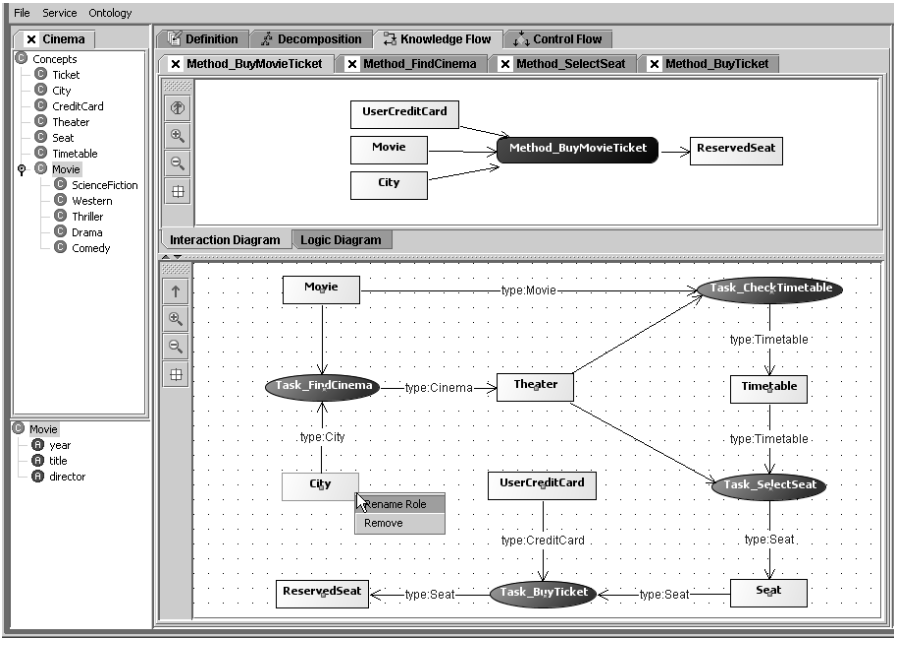

Fig. 7. Knowledge flow diagram for the composite method *Method_BuyMovieTicket* that shows the input/output interaction among the sub-tasks of the method

The knowledge flow diagram of the method *Method_BuyMovieTicket* is shown in Figure 7. The roles of this diagram are linked to the method (and domain) ontology elements, and each connection with a task has a mapping, a domain-task bridge between a domain element and a task input or output role. For example, in Figure 5 we have shown the interaction diagram of the task *Task_FindCinema*. Its output was *Theater_0*, an instance of the concept *Theater*. In the knowledge flow of the method *Method_BuyMovieTicket*, let us introduce an edge between the role *Theater* and the

task *Task_FindCinema*. In that case, as Figure 8 shows, the Mappings generator will offer the candidate list (the output role *Theater_0*) and the user could create a new interaction diagram role to map with the role *Theater*. We choose *Theater_0*, and a domain-task bridge is created. It will map the *Theater* concept of the domain ontology with the *Theater* concept of the ontology that was used to create the task definition.

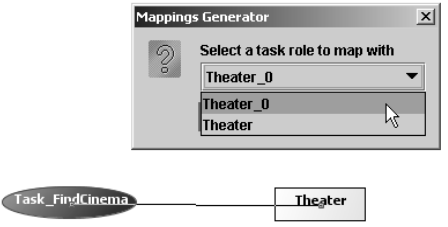

Fig. 8. The Mapping generator that presents the candidate list of possible mappings to be selected by the user

Control Flow Diagram. The control flow diagram is part of the method definition that solves the task related to the service. In this view, the user specifies the flow control of a method, where its sub-tasks are combined with programming structures to obtain a description of the service execution. In other words, this diagram describes how a composite method uses its sub-tasks in order to achieve its capability. The set of program elements that can be introduced in the control flow diagram are: (1) *condition*, depending on a logical condition, the true or false branch is chosen; (2) *conditional loop*, the true branch is executed till a determined boolean expression is false; (3) *split*, a new thread of execution is created for each output branch; and (4) *join*, synchronize or serialize various input execution tasks to one same output.

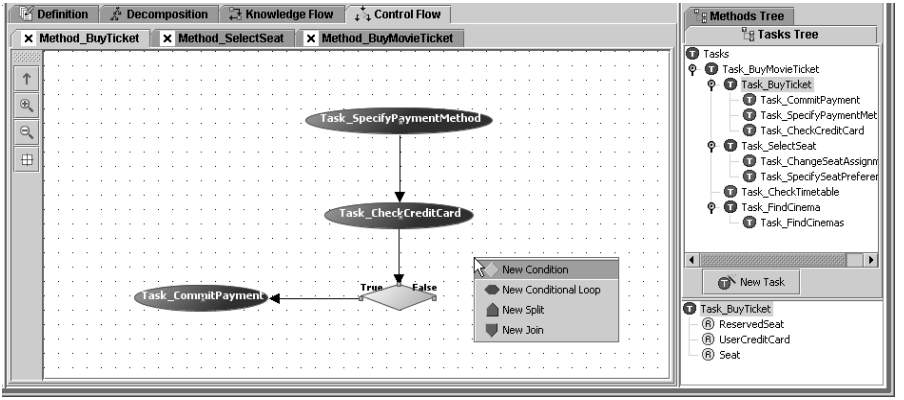

Fig. 9. Control flow diagram for the method that will solve the *Task_BuyTicket* task associated with a given service

Figure 9 shows the control flow diagram of the method *Method_BuyTicket*. First, the details of the payment are set. The next step is to check whether the credit card can afford this payment or not. If it is possible, this credit card must be charged.

Once the user has designed the service following all the complementary views provided by the SWSDesigner, it is necessary to translate that service from the graphical representation into a semantic-oriented language such as OWL-S or WSMO. To enable this translation, the SWSDesigner invokes the SWSInstanceCreator execution, which uses the graphical model of the SWS to create the instances of the SWS description ontologies. Then, the SWSTranslator is invoked to translate these instances into the language selected by the user (currently OWL-S). Figure 10 shows the OWL-S specification of the service *BuyMovieTicket* that has been automatically generated by the SWSTranslator.

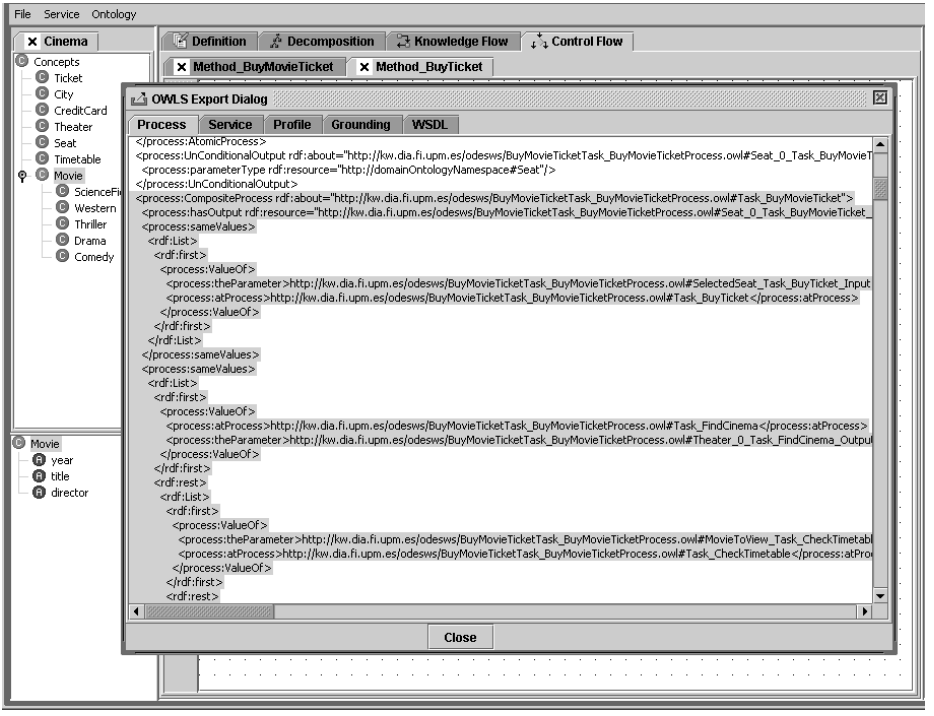

Fig. 10. OWL-S specification of the service *BuyMovieTicket*. The SWSTranslator automatically generates this specification through the SWSDesigner

4 Conclusions

Current tools that enable users to design SWSs depend on the capabilities of representation and reasoning of a specific SWS-oriented language. Those tools are con-

strained by the language expressiveness; the service must be designed using the capabilities provided by the language in which will be expressed. Furthermore, users usually introduce both errors and inconsistencies, which could be minimized using tools that operate at the knowledge level.

Consequently, to solve these problems, in this paper we claim to design SWSs at the knowledge level, and to provide tools to facilitate the design SWSs in a language-independent manner. For it, we have developed the ODE SWS conceptual framework and the environment that supports such framework. Using the graphical interface of the ODE SWS environment, called SWSDesigner, users can introduce in an easy manner the descriptive and functional features of the service, as well as, the internal components and how those components are coordinated among themselves to execute the SWS. Once the service is completely designed, and following the ODE SWS framework, it will be checked to detect inconsistencies and/or errors that could be present in the user design. If they are not detected, user will select the language (currently WSDL and OWL-S are supported) in which the SWS will be expressed.

Acknowledgements. Authors would like to thank the Esperonto project (IST-2001-34373) for their financial support in carrying out this work.

References

1. Kreger, H.: Web Services Conceptual Architecture (WSCA 1.0). IBM Software Group. http://www.ibm.com/software/solutions/webservices/pdf/WSCA.pdf (2001)
2. Curbera, F., Nagy, W.A., Weerawana, S.: Web Service: Why and How?. Proceedings of the OOPSLA-2001 Workshop on Object-Oriented Services. Tampa, Florida (2001)
3. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American. 284 (2001) 34–43
4. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems. 16 (2001) 46–53
5. Hendler, J.: Agents and the Semantic Web. IEEE Intelligent Systems. 16 (2001) 30–37
6. Dean, M., Schreiber, G. (eds.): OWL Web Ontology Language Reference. W3C Candidate Recommendation. http://www.w3c.org/TR/owl-ref (2004)
7. Sollazo, T., Handshuch, S., Staab, S., and Frank, M.: Semantic Web Service Architecture – Evolving Web Service Standards toward the Semantic Web. Proceedings of the 15th International FLAIRS Conference. Pensacola, Florida (2002)
8. The OWL Services Coalition: OWL-S 1.0 Release: Semantic Markup for Web Services. http://www.daml.org/services/owl-s/1.0/owl-s.pdf (2004)
9. Ankolenkar, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D.L., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T., Sycara, K., Zeng, H.: DAML-S: Web Service Description for the Semantic Web. Proceedings of the First International Semantic Web Conference. Sardinia, Italy (2002) 348–363
10. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Service Description Language (WSDL) 1.1. http://www.w3c.org/TR/2001/NOTE-wsdl-20010315 (2001)
11. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: Simple Object Access Protocol (SOAP) Version 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508 (2000)
12. Newell, A.: The Knowledge Level. Artificial Intelligence. 18 (1982) 87–127

13. Lausen, H., Felderer, M., Roman, D. (eds.): Web Service Modeling Ontology (WSMO) Editor. http://www.wsmo.org/2004/d9/v01 (2004)
14. Noy, N., Fergerson, R.W., and Musen, M.A.: The knowledge model of Protégé-2000: Combining interoperability and flexibility. Proceedings of the 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00). Lecture Notes in Artificial Intelligence, Vol. 1937. Juan Les Pins, Francia (2000) 17–32
15. Gómez-Pérez, A., González-Cabero, R., Lama, M.: A Framework for Design and Composition of Semantic Web Services. Proceedings of the AAAI Spring Symposium on Semantic Web Services. AAAI Press, Stanford, California (2004) 113–121
16. Benjamins, V.R., Fensel, D.: Special Issue on Problem-Solving Methods. International Journal of Human-Computer Studies. 49 (1998) 305–313
17. Motta, E.: Reusable Components for Knowledge Mdelling. IOS Press, Amsterdam, The Netherlands (1999)
18. Fensel, D., Motta, E., van Harmelen, F., Benjamins, V.R., Crubezy, M., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M.A., Plaza, E., Schreiber, G., Studer, R., Wielinga, B.: The Unified Problem-Solving Method Development Language UPML. Knowledge and Information System. 5 (2003) 81–131
19. Fensel, D.: The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods. Proceedings of the 10th Knowledge, Modeling and Management Workshop. Lecture Notes in Computer Science, Vol. 1319. Springer-Verlag, Berlin Heidelberg (1997) 97–112
20. van der Aalst, W.P., van Hee, K.: Workflow magament – Models, Methods, and Systems. MIT Press. Cambridge, Massachusetts (2002)
21. van der Aalst, W.P., ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases. 14 (2003) 5–51
22. Corcho, O., Fernández-López, M., Gómez-Pérez, A., Lama, M.: An Environment for Development of Semantic Web Services. Proceedings of the IJCAI-2003 Workshop on Ontologies and Distributed Systems. http://CEUR-ORG.com/Vol-71. Acapulco, México (2003) 13–20
23. Corcho O., Fernández-López, M., Gómez-Pérez, A., Lama, M.: ODE SWS: A Semantic Web Service Development Environment. Proceedings of the VLDB-2003 Workshop on Semantic Web and Databases. Berlin, Germany (2003) 203–216
24. Fowler, M.: Patterns of Enterprise Application Architecture. Addison Wesley (2003)
25. Arpírez, J.C., Corcho, O., Fernández-López, M., Gómez-Pérez, A.: WebODE in a Nutshell. AI Magazine. 24 (2003) 37–48