Entish: Agent Based Language for Web Service Integration

Stanislaw Ambroszkiewicz *
Institute of Informatics
University of Podlasie
al. Sienkiewicza 51, PL-08-110 Siedlce
Poland, email: sambrosz@ipipan.waw.pl

Tomasz Nowak
Institute of Computer Science
Polish Academy of Sciences
al. Ordona 21, PL-01-237 Warsaw

Abstract

A new simple minimum language Entish is proposed for automatic web service integration. The integration is done by autonomous software agents. The new language is fully declarative although it corresponds functionally to WSFL, XLANG, XAML, and DAML-S which are procedural languages. This is achieved by separating the essential data of the integration process (agent) from execution and reasoning machinery (that must be realized procedurally), and moving it outside Entish to a dedicated service (i.e., BodyService). The essential data are expressed in Entish (as agent soul) and serve as control data of the agent process responsible for web service integration. Entish is implemented on HTTP+SOAP, however, it may be also implemented on the top of SOAP+WSDL+UDDI stack if the syntax of WSDL is adopted. The idea of agent soul gives rise to introduce a new concept of agent architecture.

1 Introduction

Our project aims at creating a simple minimum language that is necessary for joining applications as web services on the one hand and for integrating and invoking them by agents (on behalf of their users) on the other hand. As this minimum we propose the language Entish, and its intended semantics.

Agent mental attributes (i.e., goal, intentions, commitments, knowledge) constitute the core of Entish. This gives rise to a new concept of agent architecture. The idea of the architecture is that agent is a temporal process created and dedicated to a particular task realization. The agent is responsible only for constructing and executing a workflow that integrates services needed for the task realization. The architecture is distributed because planning and reasoning capabilities as well as the capability to perform actions are delegated to special dedicated services that are located outside the agent.

*The work was done within the framework of KBN project No. 7 T11C 040 20.

Agent reasoning in minimized. Agent actions are restricted to communication with services and possibly migration to another host over the Internet. All the essential data of agent functioning are stored in its mental attributes. These attributes form agent soul that is separated from agent mind responsible for decision making, and from agent body responsible for action execution and environment perception.

One of the main advantage of the new architecture is that, due to the soul concept, the agent process may be closed at any time, and then reconstructed. Since the soul is independent from mind and body, it can be moved to another place and given another mind and body (i.e., agent process can be fully reconstructed) in another place. This gives rise to introduce *soul migration* as the new agent mobility form.

Since our agents are supposed to invoke web services and integrate them if it is necessary, let us start with web services.

2 Web services

What are Web services? Perhaps the best definition can be found in IBM's tutorial [10]:
Web services are self-contained, self - describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions that can be anything from simple requests to complicated business processes ... Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.
What infrastructure and what standards are necessary to realize this vision? It is clear that simplicity and ubiquity are the key factors here. From a service provider's point of view, if they can setup a web site they can join global community. From a client's point of view, if you can type, you can access services. Let us see what solutions are proposed by the prominent vendors: IBM, Microsoft, Sun, HP, and others.

Web services are getting to mean just UDDI, WSDL, and SOAP.
SOAP (Simple Object Access Protocol) is a standard for applications to exchange XML-formatted messages over HTTP. WSDL (Web Service Description Language) describes *what* a web service does, *where* it resides, and *how* to invoke it. WSDL is a general purpose XML language for describing the interface, pro-

tocol bindings and the deployment details of network services. UDDI (Universal Description, Discovery and Integration) is a standard for publishing information about web services in a global registry as well as for web service discovery.

Does the stack of standards mentioned above provide sufficient means for automatic service invocation, composition, and integration? The problem is hard. UDDI provides a mechanism for automatic service discovery of potential business partners. At the moment, it is supposed that after discovery, programmers affiliated with the business partners program their own systems to interact with the services discovered.

Automatic Web service integration requires more complex functionality than SOAP, WSDL, and UDDI can provide. The functionality includes: Transactions, workflow, negotiation, management, and security.

There are several efforts that aim at providing such functionality, for example, WSCL, WSFL, XLANG, BTP, and XAML. All these languages are based on SOAP+WSDL+UDDI stack. There is a consensus between prominent vendors that SOAP+WSDL+UDDI is the basic standard stack for automatic web service discovery. However, there is no agreement what should be the next standard in the stack necessary for automatic service composition and integration. The efforts are a basis for realizing the idea of new emerging technology: B2Bi. The current B2Bi standards initiatives include: RosettaNet, BizTalk and .NET, ebXML, Sun ONE and many others.

On the other hand there is an academic and government supported effort DAML-S. It is a part of the larger project DAML (DARPA Agent Markup Language) which aims at realizing the idea of Semantic Web. DAML-S (S for services) is based on semantic description of services.

3 Troublesome questions

The basic question is whether the proposed technologies are *simple and ubiquitous*, and which one is the right one. Perhaps the technologies are appropriate for development of serious large application. However, if a group of students and our friends wants to present their own applications as web services, invoke and compose them in an automatic way, do they have to employ all that heavy and complex machinery?

As we have seen above, the landscape of solutions for new emerging technology is rich and complex so that it is not easy to find the right path to the one common standard. It seems that the path starts with the basic stack SOAP+WSDL+UDDI, however, it is not clear how to go further. Perhaps the basic stack is not appropriate, i.e., it is too complex so that the next protocols (based on the initial stack) accumulate the initial complexity.

Are formal ontologies of services necessary for automatic web service integration ?

We are not going to give simple answers to these tedious questions. Instead, we suggest going back to the roots and asking again: What are web services ? Once we grasp the idea of web service properly, let us define a minimum protocol stack necessary and sufficient for automatic web service integration.

The following items seem to be important for web service definition:
1. A way to find and register interest in a service.
2. A transport mechanism to access a service.
3. A way to define what the input and the output parameters are for such a service.
4. A way for remote applications (services) to locate and invoke a service.
5. A way to provide transparency between users and services, i.e., a user should only formulate a task. The task performance (i.e., boring and time consuming jobs of discovering, composing, integrating, and invoking the needed services) should be done automatically.

Let us notice that there are two sides to be connected: Users who formulate tasks, and service providers who present their services on the Web. The tasks should be realized by the services. Every user should express her/his task in a language. On the other hand, service provider must express what his/her service can perform. It would be great if both the users and service providers used the same language and agreed on the meaning of that language. Moreover, service discovery, invocation, and integration should be done in the same language.

Is it necessary for the users to express the task semantics in a formal way? Is it necessary for a service provider to present a formal semantic of the operation performed by his/her service? Do we really need multiple formal ontologies and translations between them in order to realize the semantic interoperability, i.e., the agreement on the meaning of concepts used in our language? Our answer is NO. Semantic interoperability between service providers and users is realized in the course of language development, and caused by the way the language is used. Let us quote L. Wittgenstein [9]: *Don't ask what it means, bur rather how it is used.* Hence, first of all, a simple open and scalable infrastructure for language use and development should be realized. Then, the understanding (i.e., semantic interoperability) will emerge in a natural way. It seems that this is the core of the original idea of Semantic Web proposed by T. Berners-Lee in 1998.

Is it necessary to introduce actions and process constructors to the language (making the language procedural) in order to realize automatic service integration? Again, our answer is NO. Let us try to design a simple fully declarative language for web service integration by applying the Principle of Least Power (T. Berners-Lee [4]):
When expressing something, use the least powerful language you can.

4 Entish

The language is called Entish. Web service is fully described by its **name** and **the type of operation** it performs. Service name is a URI that determines the communication address as well as the transport protocol for providing communication with the service in the very similar way as it is done for URL, for

example, *hermes://ipipan.waw.pl/node/my-service* is a URI where *hermes* is a transport protocol, *ipipan.waw.pl* is the name of a host whereas */node/my-service* is the path to the service whose short name is *my-service*. Service names are elements of type *Service* in Entish.

The type of operation (performed by service) is constituted by well specified input condition and output condition, i.e., operation type is a pair of Entish formulas: *form_in* and *form_out*. The pair is an element of the type *Operation_type*. Once the formula *form_in* is satisfied, the service is invoked, the operation is performed, and the result is that the formula *form_out* is satisfied.

What does a service perform? Generally, it processes resources, that is, it performs a function on resources. Variables of that function denote input resources whereas the function value denotes the output resource. Hence, we must introduce names for functions as well as names for resource types to Entish.

There are spatial and temporal relations to be expressed in Entish, e.g., "a resource is in a service by a time". We introduce names for expressing time, i.e., the type *Time* defined according to the format **date-time** (see www.w3c.org/TR/NOTE-datetime). Since GMT time is available at any host, let us introduce the function *gmt()* that, when evaluated at a host, returns the current GMT time at the host. To express temporal relations, let us introduce the relation of the form "t1 is before (less or equal) t2", formally *(leq, t1, t2)*. For example, relation *(leq, gmt(), 2001-11-13T13:15:30)* evaluated at a place denotes that the current GMT time at that place is less or equal 2001-11-13T13:15:30. This is supposed to be the general form of timeouts for task and commitment realizations in Entish.

In order to express spatial relations let us introduce to Entish the relation symbol *is_in*, so that *(is_in, res, ser)* denotes that the resource *res* is in the service *ser*.

Let us follow the idea of T. Berners-Lee of webizing language [3], and webize Entish. Let all Entish names be URI, and let us keep Entish development open and distributed, i.e., new names for services, operation types, resources types, functions, and relations can be introduced in an open and distributed way. For this purpose, we specify distributed Dictionary-Services where everyone can create and manage his / her own dictionary consistent with the Entish syntax. The dictionary should have a form of a collection of read-only documents created according to one specific XML format.

To sum up what has been done so far, it seems that the Entish is a minimum language for service description by service providers as well as for task formulation by users. Service invocation is realized by satisfying the formula *form_in* of the type of operation the service performs. The idea of service invocation appears to be great, however, the question is: Who or what is responsible for realizing this satisfaction. Who or what is responsible for realizing service composition and integration if it is needed? The answer is *agent*, i.e., there must be a process (called agent) responsible for task realization. Hence, we introduce the type *Agent*.

For any task issued by a user there must be an agent responsible for the task realization. In order to do so a user needs a GUI that is called SecretaryService. For any task there is a timeout for its realization. The task with a timeout is written down as the agent's goal. The agent is dedicated only to its goal realization, so that when it succeeded or the timeout is over, the agent notifies the SecretaryService about that, and terminates its process. Hence, we introduce to Entish the first agent attribute: *goals(agent)*.

What is the agent supposed to do after receiving the goal? It starts with its main goal as its first intention. It asks any service by sending the following message: "My intention is ϕ", where ϕ is an Entish formula describing the goal of the agent. Hence, we introduce to Entish the next agent attribute *intentions(agent)*. If the service is able to realize the agent intention, it replies to the agents with a commitment that has the following form: "I commit to realize your intention, if the conditions *con1 and con2* are satisfied."
These conditions describe the input resources the agent is obliged to deliver to the service as well the timeout for the delivery. Hence, we introduce to Entish the attribute *commitments(service)*, that is a pair of *form_in*, and *form_out* formulas, where *form_in* is equal to *(con1 and con2)* whereas *form_out* is equal to *intentions(agent)*. The information about the commitment must be saved in agent's knowledge, so that we introduce to Entish the next agent attribute: *knows(agent)*. Usually, agent is equipped with initial knowledge by the SercretaryService. Agent can also commit to a service (particularly to its SecretaryService) to perform a task, so that we introduce also the following agent attribute: *commitments(agent)*.

Service invocation is realized in the following six steps: In the first step an agent sends its intention to the service. In the second step a commitment is sent to the agent by the service. In the third step the formula *form_in* of the commitment is satisfied by the agent or by another service. In the fourth step an operation is performed by the service. In the fifth step the formula *form_out* of the commitment is satisfied. Let us note that there may occur failures in the steps: second, third, and fourth one if, for example, the timeout is over. In the final sixth step the agent is notified by the service about either success or failure. The service invocation constitutes the first crucial point of our approach.

Generally, if agent's goal can not be realized by a single service, the agent looks for a plan (called workflow plan) that decomposes the main goal into sub goals (sub tasks). Once it has got a plan, it must find out services that can realize the sub tasks, and arrange the workflow. How can it be done? The simplest solution is to introduce a special service (called InfoService) for providing agents with workflow plans and info about services performing operations needed for the workflow. However, the workflow arrangement, execution and control, as well as reconfiguring and re-

covery in the case of failure is delegated to agents. Workflow plan is an element of type *Operation* in Entish. It is a sequence of Entish formulas to be adopted by agent as its intentions. The last formula of the sequence describes the final result of the workflow execution, whereas the first formula describes what initial resources are needed to start an execution. The rest of the formulas describe intermediate situations and correspond to operations to be performed in the workflow.

The way the agent constructs a workflow on the basis of a fixed workflow plan is the following. The agent starts with the last formula in the workflow plan sequence as its first intention. It is supposed that the agent's goal follows from that last formula. The service **SER-0** that agrees conditionally to realize the first intention, gives to the agent an appropriate commitment where the *form_in* formula of the commitment determines (together with the work plan) the next intentions of the agent. The next move of the agent is to find out service(s) that can realize the next intentions. Once the agent finds out an appropriate service, say **SER-1**, that commits to realize the intentions, the satisfaction of the *form_in* formula of the service **SER-0** is delegated to the service **SER-1**. (**It is the critical point for understanding our idea of automatic service integration.**) Supposing that the agent finds out the service **SER-1** that agrees to realize its intention, the service **SER-1** returns to the agent a commitment with another *form_in* formula that determines the next intentions of the agent. Once the agent finds out an appropriate service, say **SER-2**, that commits to realize the intentions, the satisfaction of the *form_in* formula of the service **SER-1** is delegated to the service **SER-2**. And so on. The process goes on until the agent collects all commitments needed to construct a workflow according to the adopted plan. The commitments include appropriate timeouts that synchronize the workflow. Once the agent computes (on the basis of its knowledge) that it itself can satisfy initial formulas in the workflow plan, it can start the workflow execution. Usually, the initial formulas correspond to the initial resources (data) provided by the SecretaryService for the task realization. The satisfaction of the initial formulas starts the workflow execution; it looks like domino effect. If the process is completed before the timeout set for the task realization, the agent can execute the workflow, and then notifies the SecretaryService about that. The method of constructing and executing workflows by the agent constitutes the second crucial point of our approach.

InfoService is crucial for task realization by the agents. From the point of view of system functioning it is a distributed database, and it is not important how it is implemented. The only requirement is that InfoService implements simple conversation protocol for providing info (expressed in Entish) about operations (workflow plans), and about services having specified operation type. This is done in response to agent's request that is always of the form: "My intention is ϕ". InfoService is also used for publishing info by service providers. Service provider can send an info (as Entish formula), describing the type of operation his/her service provides, to InfoService. InfoService can joint this info to its database. Hence, the conversation protocol for InfoService is extremely simple. InfoService corresponds to UDDI, however, it seems to be simpler. Agent is dedicated only to one task performance, but its experience is supposed to be saved in an InfoService where it is processed and the results help other agents that have tasks of the same type. The idea of InfoService constitutes the third crucial point of our approach to service integration.

The idea presented above seems to be nice, but the problem is how to create an agent that could perform tasks by arranging and executing workflows. How should such agent react to failures of some workflow elements? What if the agent process is killed by an accident or if the host of the agent process is down? The solution is quite simple: The essential data of agent process must be separated from execution machinery of the process. Surprisingly, these data correspond to agent (mental) attributes. Let these data be called `soul` in Entish. The execution machinery should be delegated to the special dedicated service called BodyService. Agent process is created if the data structure `soul` is sent to a BodyService. BodyService is responsible for agent's reasoning, planing, communication, and action execution on the basis of the soul data. If the process is killed by an accident, then it can be fully reconstructed from the `soul`, because the `soul` stores (by definition) all essential data of the process. Since the `soul` is expressed in Entish, it is implementation independent. This constitutes the fourth crucial point of our approach.

To sum up, Entish is fully declarative language although it corresponds functionally to WSFL, XLANG, XAML, and DAML-S which are procedural languages. To be more precise, Entish (being declarative) supports automatic service integration which is also the intended goal of WSFL, XLANG, XAML, and DAML-S. It is the concept of separation of the essential data (i.e., soul) of the agent process from the process execution and reasoning machinery that allows to keep Entish as a declarative language. The essential data are expressed in Entish and serve as control data of the service integration process. The execution and reasoning machinery (that must be realized procedurally) is moved outside Entish to the dedicated service, i.e., BodyService. BodyService is viewed as an application implementing the appropriate interface for communication in Entish. However, we impose several conditions on the BodyService behavior, for example, agent's goals and commitments can not be canceled unless the associated timeouts are over, an agent can not take intentions that contradict its goal or commitments, and finally agent must be rational whatever it means. These requirements are not formal but are necessary for assuring intended system behavior.

Of course, Entish could be extended by introducing actions and process constructors, so that execution, reasoning and control could be described. However, the design motto was: *When expressing something,*

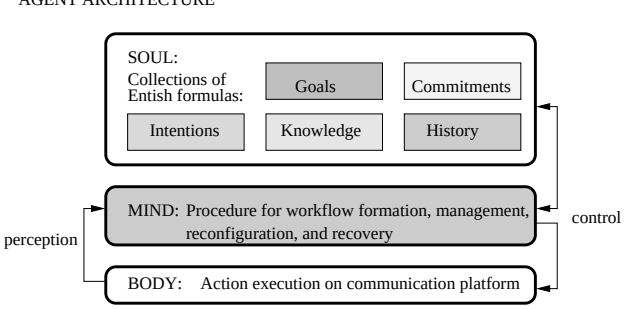

Figure 1: The layered agent architecture.

use the least powerful language you can. Hence, we have designed a minimum language. What was unnecessary was moved outside, that is, the functionality associated with service publishing and discovery was moved to InfoService, whereas execution and reasoning machinery was moved to BodyService. As a result, Entish is a simple declarative and action independent language for web service integration.

Let us compare Entish to the existing web service integration efforts. The point of inventing Entish was not to create a new better language for service description. Entish as a web service description language is simple, perhaps too simple so that WSDL should rather be adopted as the basis for developing Entish idea. The novelty the Entish proposes is a new technique for automatic service invocation and service integration. It corresponds to the efforts of WSFL, XLANG, XAML, and DAML-S. Hence, the idea of Entish could be applied to built the layer on the top of SOAP+WSDL+UDDI stack. In order to do so the syntax of WSDL should be adopted. However, the syntax of Entish seems to be easier for presentation as well as for implementing. For this very reason we decided to adopt only SOAP as the transport layer, and to implement Entish on SOAP.

5 Agent architecture

We propose the following agent architecture, see Fig. 1.

Agent architecture is composed of the following three layers: mental attitudes (soul), decision mechanism (mind), execution mechanism (body). The mind and body is implemented as a BodyService although functionally they are different modules, and they should be separated. The mind performs reasoning and planning on the basis of the soul contents and the perception delivered from the body. The mind controls both the soul and body. The body executes actions (directed by the mind) in a transport platform, e.g., on SOAP+HTTP.

The soul is expressed in Entish and consists of knowledge (a collection of facts), goals, intentions, history and commitments. Since the contents of soul is expressed in Entish, it (the contents) is independent from transport platform as well as from agent implementation. Agent soul can migrate alone without mind and body. This gives rise to introduce new form of agent mobility called *soul migration*. The idea is that a running agent process stores all its essential data and control parameters in its soul. The process may be closed at any time, the soul sent to a BodyService running on a remote host, and the agent process can be fully reconstructed there. The new agent migration form is based on one common format of agent data (soul). Usually, agent data structure is arbitrary and depends on agent implementation, so that migration of agent data makes sense only if the same agent code is available at the new place. In our approach agent data has one universal structure so that it is compatible with any agent code that implements the common format of soul.

Although the soul is based on the idea of BDI agent architecture [5; 7], the proposed soul format may be insufficient, that is, some important aspects (data) of agent process are not stored in the soul. The proposed soul format was design for agents that are supposed to integrate web services. So that from this point of view it works well. Perhaps the format is not universal so that in order to apply our agent architecture in other domains, the format should be extended.

A consequence of our agent architecture is that agent is purely an information agent. Its capability to perform actions is restricted mainly to migration and communication however, there is a possibility of having more actions implemented as routines in agent body. Its ability to reason is minimal, almost all reasoning and planning job is delegated to the InfoServices.

So, the question is what our agent does? The main and only job of the agent is to construct workflow that would perform successfully the task delegated to that agent. This includes: (1) construction of operation (a sequence of sub tasks) of the workflow; (2) arrangement of services needed to perform the sub tasks; (3) control, rearrangement and recovery of the workflow in the case of failure; (4) and finally notification (either positive or negative) of the task realization to be sent to user.

Agent decision mechanism can work according to the following algorithm:

1. update knowledge on the basis of perception;
   check all your timeouts;
2. check if your current intention and/or goal is realized;
   if the goal is realized or the timeout to realize it is over, go to (7);
   if the current workflow execution fails, go to (6);
   if the current intention is realized go to (6);
3. check if there is a routine (a primitive action) in your body that can realize your current intention;
   if there is one, execute it and go to (1);
4. ask a service how to realize the intention;
   if you get back a partial plan, go to (6);
   if you get back a commitment, take the precondition of the commitment as the current intention;
   if the precondition is true, go to (5), otherwise go to (1);

if all known InfoServices can not help you, migrate randomly to another remote place (to look for another InfoServices) and go to (1);

5. if the workflow is completed, execute it and go to (1);
6. planning and determining (next) current intention;
   go to (1);
7. notify (positively or negatively) the user;
   give a report to an InfoService;
   terminate the process.

6 Conclusions

It seems that our agent architecture can not be classified according to the standard taxonomies, see for example [6; 8]. Perhaps it is specific for application domain, i.e., web service integration. Our architecture may be seen as distributed, i.e., InfoServices may be viewed as part of agent architecture where the main part of agent planning and learning is performed. The learning consists on storing and processing agent experience. Agent life is short, i.e., it is dedicated to exactly one task performance. It "dies" after realizing its task (or if the timeout is over) and reporting the way the task was achieved to an InfoService. On the other hand, it is easy to create a new agent if there is another task to be realized. The new agent can use the experience (of the past agents) stored and processed in the InfoServices.

A detailed syntax of Entish with explanation and walk trough example are available on request from `sambrosz@ipipan.waw.pl`
The first prototype of Entish implementation was done on HTTP+SOAP transport. Simple instances of SecretaryService, InfoService, DictionaryService, BodyService and a number of ordinary services were implemented. Most of the ordinary services are for converting data formats, e.g., gif to jpg, pdf to ps, latex to html, etc.. There are also other services like PhoneNumberService that returns the phone number of a person given his/her personal data (name, address), and a lot more will be implemented as student projects shortly. Now, we are testing the prototype and collecting experience. One corollary is obvious: More services are needed.

A preliminary version of Entish has been published in [1]. For sources and reports on the progress of Entish specification and implementation, see our web site: `www.ipipan.waw.pl/mas`

References

[1] S. Ambroszkiewicz and T. Nowak. Agentspace as a Middleware for Service Integration. In Proc. ESAW'2001. Springer-Verlag LNAI, vol. 2203.

[2] S. Ambroszkiewicz, W. Penczek, and T. Nowak. Towards Formal Specification and Verification in Cyberspace. Presented at Goddard Workshop on Formal Approaches to Agent-Based Systems, 5 - 7 April 2000, NASA Goddard Space Flight Center, Greenbelt, Maryland, USA. Published in Springer LNAI Vol. 1871.

[3] T. Berners-Lee - www.w3.org/DesignIssues/Webize.html -and- /DesignIssues/Logic.html

[4] The Principle of Least Power (T. Berners-Lee) www.w3.org/DesignIssues/Evolution.html

[5] M. E. Bratman. Intentions, Plans, and Practical Reason. Harvard University Press, 1987.

[6] J.P.Mueller. The Right Agent (Architecture) to Do the Right Thing. In J.P. Mueller, M.P. Singh, and A.S. Rao (Eds.) *Inteligent Agents V, Proc. of ATAL'98,* Springer LNAI 1555, pp. 211-225, 1999.

[7] A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI–architecture. In Proc. KR'91, pp. 473-484, Cambridge, Mass., 1991, Morgan Kaufmann.

[8] W. Truszkowski and J. Karlin. A Cybernetic Approach to the Modeling of Agent Communities. In Proc. of 4th International workshop CIA 2000, Boston, MA, USA, July 2000. LNAI 1860, pp. 166-178.

[9] L. Wittgenstein. *Philosophical Investigations.* Basil Blackwell, pp. 20–21, 1958.

[10] IBM's tutorial www-4.ibm.com/software/solutions/webservices/

[11] DAML-S www.daml.org/services

[12] OASIS BTP www.oasis-open.org/committees/business-transactions/

[13] UDDI www.uddi.org

[14] SOAP and XMLP www.w3.org/2000/xp/

[15] XAML www.xaml.org

[16] XLANG www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[17] ebXML www.ebxml.org/

[18] WSFL www-4.ibm.com/software/solutions/webservices/