Virtual Web Services

Jarogniew Rykowski
Department of Information Technology
The Poznan University of Economics
Mansfelda 4, 60-854 Poznan, Poland
tel./fax +48 0-61 848-0549
e-mail: rykowski@kti.ae.poznan.pl

ABSTRACT

In this paper we propose an application of software agents to provide *Virtual Web Services.* A *Virtual Web Service* is a linked collection of several real and/or virtual Web Services, and public and private agents, accessed by the user in the same way as a single real Web Service. A Virtual Web Service allows unrestricted comparison, information merging, pipelining, etc., of data coming from different sources and in different forms. Detailed architecture and functionality of a single Virtual Web Service is user-dependent, and information gathered from existing *Web Services* may be used in an individual manner. The main goal of the proposal is twofold. First, virtual services allow unrestricted personalization of any Web Service by user-defined software executed at both the server- and the client-side. Second, virtual services provide efficient server-side monitoring and alerting once "vital" information provided by a real service is changed, and this change is of any interest to particular user. In addition, the service users are able to define their own, non-standard interfaces to existing services without a direct interaction with the service provider (information owner). This feature allows for user-specific versioning of services and continuous improvement of the service from the user point of view. By shifting the personalization aspects to the users, we reduce overall maintenance costs (from the service owner point of view) and improve system flexibility and fast adaptation to dynamic changes in the environment and evolving user requirements.

Keywords
Web services, software agents, personalization, and customization

1. INTRODUCTION

Recently, we may observe two basic trends in the worldwide economy: the globalization, and widening use of new information and telecommunication (IT) technologies. These trends complement each other, having a lot of influence over current business methods and contacts. Traditional ways of doing business are now being replaced by modern solutions based on efficient information flow in a global, distributed environment. The main barrier to the worldwide mass deployment of modern e-business solutions is a lack of interoperability of different software systems used by business partners. Hence, a research interest is observed in such systems as Web Services, aiming at overcoming the non-compatibility and non-operability restrictions. The general idea of Web Services is to provide an additional standardizing layer for linking e-business applications in a uniform, worldwide manner [SOA, 2004]. In a typical Web Service application, the way of data interchange is standardized, as well as (to some extent) data semantics and service functionality [Microsoft's Web Services, 2005].

Web Services were originally proposed for the traditional-business application areas, with stable and long-lasting relations among the business partners. Thus, this technology is focused on providing stable and universal services, to be accessed as needed. Possible interaction with a service is determined in advance by the service owner, and information format and semantics is fixed. A well-defined set of given Web Services may create service conglomerate, either orchestrated, or choreographed to achieve common business goals. The primary difference between the orchestration and the choreography is a scope. The choreography model provides a larger scope, encompassing all business parties and their associated interactions (e.g., a peer to peer model). The orchestration model concerns mainly relations between two parties (specifically focusing on the view of one participant) [BPEL, 2006; WS-CDL, 2006].

One of the main problems of efficient orchestration and choreography of Web Services is related with an adjustment of service interface and functionality to the needs of the service clients, i.e., the business partners that are interested in cooperation with the service. Several problems were discovered related with such service personalization and optimization from the client point of view, in particular: (1) an impossibility of foreseeing all the possible expectations of all the possible clients that implies an impossibility of providing a stable long-term service interface, common for everybody, (2) an impossibility of taking into account some specific features of the access method, such as limitations of the communication channel, client-related security aspects, etc., (3) limited possibility of server-side monitoring of changes of the information provided by a service, (4) limited flexibility and slow and costly adaptation to new business conditions, and (6) limited support for ad-hoc and temporal business contacts [Rykowski, 2006-D]. These problems are further discussed in Section 2 of the paper.

In order to overcome the above-mentioned problems, in this paper we propose virtual services to assure effective service personalization, grouping, pipelining, change monitoring and alerting, fast adaptation to dynamic business requirements, and support for ad-hoc business collaboration. A *Virtual Web Service* VWS is a linked collection of several virtual and/or real services, accessed by the user in the same way as a single service. Each VWS is created for (and possibly by) a particular user, so it may be personalized to the maximum extent. A VWS may act a wrapper to real Web Services, allowing their personalization. A VWS may combine an arbitrary number of different services, allowing comparison, information merging, pipelining, etc., of data coming from different sources and in different forms.

To successfully implement the Virtual Web Services idea, we need a technology capable of efficient execution of autonomous, distributed, and possibly mobile programs. Such programs should be prepared and controlled by different users (business partners), and commonly orchestrated to achieve case-specific business goals under different restrictions. We propose to implement virtual services by the use of the software agent technology. The main goal of applying software agents for Virtual Web Services is twofold: increasing the level of personalization of individual Web Services, and allowing a combination of several independent Web Services into a single, consistent service. To this end, the agents are used as information brokers and wrappers to existing Web Services, to adjust data format (both syntax and semantics) to the specificity of the business partners – service clients. The brokerage/wrapping algorithm may be programmed by any business party – a client, a service provider, or both. The agents may also act as monitors and asynchronous notifiers about important data changes in inspected Web Services. What is "important" is programmed in agent code and variables. The agents may interact with local and remote software systems, Web Services included, as well as with other agents, and with humans. The agents are executed in the scope of the Agent Computing Environment (ACE) framework [Rykowski, 2003-B; Rykowski, Juszkiewicz, 2003]. ACE framework consists of a set of Agent Servers [Rykowski, 2005-A] located at selected network hosts. The agents may be moved among Agent Servers, according to the current situation and requirements of business partners. In particular, an ACE agent may be executed at client-side (i.e., in client's local area network), at server-side (i.e., in a network belonging to a service provider), or at an external network host (i.e., on a separate host outside clients' and providers' networks).

The remainder of the paper is organized as follows. In Section 2, basic ideas of Web Services are presented, and some basic limitations of this technology are discussed, widening the debate from the Introduction. In Section 3, agent technology is briefly described, including the ACE agent-based framework for creating and executing user-defined software agents. Next, overall system architecture is described, and some implementation issues are discussed related with the adaptation of the ACE technology for the personalization of Web Services. In Section 4, a comparison is included of the presented work with similar proposals, covering both Web Services and, more general, software agent technology. Section 5 concludes the paper.

2. WEB SERVICES

Web Services WS [Web Services, 2002; Lombardi, 2002] are encapsulated, self-descriptive, modular, Internet applications that may be accessible by the users via the network. Their basic purpose is to enable standardized, uniform access to heterogeneous, distributed software, running on different software/hardware platforms. Generally, a Web service:

- is a software application, accessible via standard communication protocols (HTTP, XML, SOAP, etc.),

- may be accessible via firewall and proxy connections,
- is integrated, by the use of XML documents, with other software applications and systems, using different communication and data representation standards (RDBMSs, CORBA, COM, etc.),
- is platform-, access-, and client-independent.

Web Services are not implementations of particular services. They are rather an encapsulation layer for the software they are linked to, providing standardized ways of data interchange, searching mechanisms, and interface description. Due to such wrapping, it is possible to use a functionality of a remote application without detailed knowledge of its structure and API. Thus, Web Services are capable to ensure interpretability of different applications, prepared and so far used independently of each other.

Web Services are not fully standardized yet. They are under continuous development, especially by large companies like IBM [SOA, 2004] and Microsoft [Microsoft's Web Services, 2005].

Below, a discussion is included related with several aspects of efficient personalization of Web Services and the usage of Virtual Web Services, namely:

- restricted individualization of a service according to different requirements of end-users (business partners), including a choice of data format, contents, place and time of the delivery, etc.;
- some independent restrictions related with hardware, software, and communication means, used by the end-users to access given services, e.g., small screen and keyboard of a mobile phone, expensive WAP/GPRS [Bettstetter, 2002] connection, frequent disconnections from a mobile network, etc.;
- limited ways of user-defined composition and comparison of several Web Services, offered by different business partners; the services offered by different partners are optimized from the service provider point of view; moreover, different business partners are usually not interested in global standardization of their offers, for both technical and economical reasons;
- limited possibility of server-side monitoring of changes of the information provided by a service;
- high costs of adjusting Web Services to the detailed specificity of potentially many business partners, usually unknown in advance,;
- limited flexibility and slow and costly adaptation to new business conditions, and
- limited support for ad-hoc and temporal business contacts.

These problems are briefly discussed below.

2.1 Restricted customization of Web Services

A need for service customization is a natural trend in a global, distributed system, with the end-users coming from different countries, cultures, time zones, legal systems, business traditions, habits, etc. Each service user tends to fit the service into his/her individual requirements and expectations. However, service providers are usually not interested in individual adaptation of their services according to detailed requirements of particular end-users. There are two main reasons for that. First, it is impossible to foresee all the possible expectations of all the possible users. Even if all current users are satisfied with the service, no one can guarantee that a new-coming user is not asking for a change. Moreover, expectations of different users may be conflicting or even contradictory. Second, increasing service complexity is followed by the increase of costs of service development, and seriously complicates service usage, especially from the point of view of undemanding users. As a consequence, a reasonable trade-off must be found between service complexity and abundance of different users' requirements, optimized from the service provider point of view. Thus, only a limited functionality for a service is usually provided, and unsatisfied users must perform additional data processing at user-side. As a consequence, each user has a choice to: either (1) slightly personalize the service functionality/data at server-side, using for example user profiles, custom functions, cookies, etc., or (2) fetch all the data from the service and perform additional operations at client-side. In the first case, the user has limited possibilities of adjusting the service, because of limited number of parameters available in user profile. In the second case, data flow increases significantly though unnecessarily, because usually most of the transmitted data is filtered out at the client-side. Moreover, end-user device must be powerful enough to provide complex computations, which is not the case of many light devices like mobile phones and PDAs. Thus, neither the first nor the second solution is sufficient for many users.

Limitations of the level of service customization are not only related to the above-described technical reasons. Each service is optimized from the service provider point of view, and the service owner tries to reduce costs of service development and maintenance. Service personalization, as not crucial from the service owner point of view, is a first-order candidate for such reduction. As a consequence, due to both hardware and software limitations, and costs of preparation and use of the service, personalization is typically restricted to some simple built-in mechanisms, as mentioned above – mainly server-side-maintained user profiles, preferences, dynamic cookies, sessions, etc. All these personalization mechanisms are internal parts of the service. As such, they cannot be changed by the end-users. As a consequence, from the end-user point of view the level of personalization of the service is often unsatisfactory.

For example, consider a factory producing and selling car parts. A name of the part as well as its description is usually different for the business partners involved in the cooperation. For example, “wheel model W1234” of the part producer becomes “standard wheel” for the car-assembly factory. Also, the financial aspects of the business process of selling and buying the car parts are maintained by two different computer systems – the way if identification of the invoices, bank accounts, local taxes and other financial duties is different for the business partners and should be mapped during the transaction to uniquely identify the financial flow. For example, the invoice “1/2006” issued by the part company is stored as “999/06/2006” incoming invoice in the internal financial system for the car-assembly factory.

Beside the above human-related personalization, one may consider several user-independent limitations related with end-user hardware, communication channel throughput, costs of a connection and data transfer, dealing with disconnections and other communication errors, and many more. These limitations are especially important in the scope of a global system connected via modern, mobile telecommunication facilities – GSM/UMTS phones, WiFi/WiMAX equipment, etc. It is difficult and economically unjustified to provide a single universal service dealing with any kind of end-user device connected via any communication channel. This assumption strongly limits a possibility of accessing a service from anywhere and anytime, which was one of the key issues for the introduction of Web Services. However, these limitations may be bypassed on condition the auto-adjustment to end-user device and communication link is performed outside the service, by the service clients.

Little work so far has been devoted to the customization of Web Services. In most of the proposals, standard mechanisms already developed for Web servers are adapted for Web Services [Bonett, 2001]. These mechanisms are usually based on server-side user profiles, and client-side cookies. To build a user profile, two approaches are used: a static and a dynamic one. In the static approach, this is the user who is responsible for manual filling his/her profile, by the use of different questionnaires, forms, interviews, etc. The questionnaire data is analyzed statistically, and the user is assigned to a predefined user profile (e.g., user profiling for Yahoo public Internet searching engine). In the dynamic approach, all the activities of the user are analyzed. Basing on these activities, the user is assigned to one of the predefined profiles. The user activities are continuously monitored, thus the assignment to the user profile may vary in time. There are several techniques for activity monitoring: Click-stream Analysis, Web Usage Mining Systems, Collaborative Filtering, cookie analysis [Bonett, 2001], etc.

The main problem in using the above customization method is a determination of useful set of predefined user profiles. A number of these profiles must be reasonably small; however, this limits personalization scope. More the personalization is group-aware rather than user-aware, less the users are satisfied [Lombardi, 2002; iMesh Toolkit, 2003].

In 2002 IBM released the Web Services Experience Language WSXL, an XML-based description language that offers methods for creating interfaces, accepting user input, and distributing these interfaces. It has facilities for displaying branding information and integrating with many other XML-based standards [WSXL, 2002]. Mozilla's method of interface customization is another experimental option. Mozilla uses XUL [Deakin, 2004], an XML-based user interface language, to define its own user interface. Using XML, CSS, JavaScript, and RDF, one can make full application interfaces – not just Web pages – and experience the same kind of process one might have when designing a user interface for Web services [AMASE, 2002].

Recently, some systems have been proposed using software agents as brokers to Web Services and similar information sources, e.g., the system proposed by Pinsdorf et al. [Pinsdorf, 2002] based on the SEMOA platform. In the proposal, mobile software agents are used as proactive components, realizing interconnections between data sources and the users. A specific agent is chosen based on the user's query type. If a query has to be performed, the according agent is instantiated, and migrated to a number of data sources. By the end of its work, the agent composes and returns a report that aggregates the results of all agent's queries to the information sources. Similar to our Virtual Web Services, the proposal is based on the idea of an interaction of a single agent with a well defined, rather small set of information sources. Thus, it is sufficient for each agent to know the structure of the small number of specific data sources – the process of the development of agents is relatively simple. The main difference of the above-mentioned proposals and our Virtual Web Services lies in the fact that the agents (together with the whole service composition) are created and managed by the end-users – agent owners. Thus, our proposal is much more flexible and better adjusted to specific needs of particular users, while in [Pinsdorf, 2002] fixed types of agents are used related with query types rather than end-user specificity.

2.2 Restricted composition and comparison of Web Services

In business practice, often information coming from different sources has to be compared and merged. End-users often tend to work with several competitive Web Services at a time, e.g., to compare the costs and profits of different service implementations, to choose the most adequate service functionality, to change the service provider when overall situation is changing, etc. Another example is service pipelining, where information being a result of one service is used as an input for another service. Thus, a need for personalized, usually dynamic orchestration of different services arises [Rykowski, 2006-C]. The problem is similar to the personalization of a single service described above, except that (1) usually it is impossible to provide links among services at server-side, moreover, service providers are usually not interested in linking to the services of possible competitors and business rivals, (2) services are not fully compatible at the functionality/semantics level, so separated data processing and wrapping is needed, and (3) an amount of information exchanged may be quite large. One may suppose that recently proposed extensions such as Universal Description, Discovery, and Integration UDDI [UDDI, 2005] may help to solve this problem. However, functionality of UDDI and similar catalogues is not sufficient, as these catalogues are used for searching rather than comparing and combining the services. As a consequence, to create a useful group of combined Web Services, end-users must directly fetch data from many services and process all the obtained information at the client-side.

Recently, two proposals have been published for efficient orchestration of Web Services, namely BPEL language, and WS-CDL language. Business Process Execution Language BPEL [BPEL, 2006] grew out of Microsoft's WSFL [WSFL, 2005] and IBM's XLANG [XLANG, 2005]. The language is serialized in XML and aims to enable programming in the large, referring to the high-level state transition interactions of a business process – BPEL refers to this concept as an Abstract Process. A BPEL Abstract Process represents a set of publicly observable behaviors in a standardized fashion. An Abstract Process includes information such as when to wait for messages, when to send messages, when to compensate for failed transactions, etc. Define business processes that interact with external entities through Web Service operations defined using WSDL 1.1 [WSDL, 2001], and that manifest themselves as Web services defined using WSDL 1.1. As the BPEL language is based on the technologies already widely used for the Web Services (XML, WSDL), the application domain of this proposal does not cover, similar to the Web Services, dynamically evolved and ad-hoc business activities.

Web Services Choreography Description Language (WS-CDL) 1.0 [WS-CDL, 2006] is a similar proposal, however, addressed to the problem of choreography rather than orchestration of services (cf. Section 6.1). WS-CDL describes and sets rules for how different Web services components will interact, according to the W3C. It provides tools for sequencing and offers a "flexible systemic view of the process". The W3C positions WS-CDL "as a necessary complement to BPEL, Java, and other programming languages which describe one endpoint on a transaction, rather than the full system". This project is in the early stage of development, however, and so far only the first draft of the language specification has been published.

Returning to the example from the beginning of Section 2, imagine now that the car-assembly factory mentioned in the example is supported by several companies producing the parts of the same functionality and quality. Thus, a single service would be useful to choose the most adequate business partner and its Web Service at the moment. To this end, Virtual Web Services may be used with automatic detection of the optimal (under current and possibly dynamic conditions) business offer. Basic functionality of such virtual service is to dynamically compare all the offers and to choose the best (from the point of view of the service owner, i.e., car-assembly factory) offer.

2.3 Restricted server-side monitoring of changes

In many cases what is really important is not the fact of possessing the information, but the fact of detecting a change of this information. In a classical approach to implement a Web Service, when the service passively waits for user demands, these are the service clients who must periodically and manually poll for new information to compare it with the previous one and detect information changes. Such approach generates big network traffic and consumes client and server resources. Thus, polling for information should be replaced by pushing information changes by the service to the users, i.e., making the service active. Some techniques have been proposed towards this goal, mainly in the domain of databases, namely, database triggers, and active databases. Database triggers are usually implemented as a set of procedures auto-executed by the database management system once a given event occurs. Triggers are widely used as a mechanism for on-line verification of database integrity [Oracle Triggers, 2006]. Triggers may be used for personalization; however, as they were not designed to this goal, such personalization is restricted. First, triggers are common for all the database users. As such, triggers are usually made by system designers, while users cannot personalize them. Second, triggers are connected with a given database table rather than a user or a query. Third, triggers cannot modify query results. Fourth, current trigger implementations are limited to "insert-update-delete" commands only – application of triggers to read-only queries is not implemented in current database management systems.

Active databases [Ceri, 1997; Paton, 1999] are, in some sense, an extension of the triggering mechanism (in a sense of a concept; not as a historical successor). In active databases, an automatic system reaction is programmed by a set of mutually connected *event-condition-action ECA* rules. The idea of active databases [ACT-Net, 1996] seems to be abandoned, mainly due to impossibility of development of efficient methods of database/rules management. The prototypes never reached the commercial status, and a lot of basic functionality has never been implemented, including an efficient debugger for rules and actions triggered, query optimizer specialized for ECA rules, and multi-user engine. These facts, together with the technical difficulties of individualizing the set of rules for different users, makes the concept of active databases practically useless in the scope of personal information monitoring.

Efficient, user-specific monitoring of changes of the information provided by a Web Service would be quite useful, with some alerts generated by server-side close to the information source. For the reasons described earlier in this section, it is quite difficult to obtain such functionality at the server-side. In turn, providing monitoring at the client-side increases the amount of data transmitted to and from the service, thus client-side monitoring is not acceptable for most of clients. Moreover, continuous monitoring forces periodical, massive access to a service by many users that in turn may substantially reduce throughput of the communication channel to the service and response time for other users. Unfortunately, to apply user-defined monitoring and alerting, current implementations of Web Services require on-line continuous connection with the service and change detection at client-side.

2.4 Restricted support for ad-hoc and temporal business activities

As mentioned above, Web Services are not yet stable and standardized enough. As such, the technology is not accompanied with some additional tools to cut costs and facilitate the process of development of new services. Relatively high costs of preparation of a single service force the service provider to plan using the service in a stable manner for a long time, with many clients, even if the overall situation is changing rapidly. Thus, a typical Web Service is not flexible enough (although it could be according to the possibilities of the technology) to deal with evolving requirements and expectations of the users, as well as an evolution of the environment (hardware, software, overall business relations, etc.). Including the changes while keeping the service functionality accessible for the users for a longer time in the same manner usually forces the service to be too complex and too parameterized for most of the users. As a consequence, either the service is unchanged and stable for a long time, despite the changes in the environment and evolving user expectations, or the service is too complex and too costly for both the service provider and the end-users.

There is another issue of the Web Services that may result in decreasing the application scope of this technology. In many business cases, especially during the transitional period between the old and the new economy, a need arises to support some business activities that cannot be fully automated. However, the basic assumption for a typical Web Service is that there is no place for human work, and the business relations are performed only via computers and the computer network. Even if the human work is needed for a given service, such work is realized as back-office activities, of no interest to the service clients. Thus, applying Web Services to partially manual collaboration would be difficult and costly. Similar, fast prototyping of automation of some ad-hoc business relations is not possible in the scope of this technology, as the services must be applied at once, with no way of supporting these relations partially.

A continuous process of service improvement, typical for many business relations, is quite hard to be automated by the use of Web Services. On the contrary, for a typical Web Service, a common assumption is made that all the details of the business process are known in advance, before service implementation. Recently, a lot of work is devoted to keeping consistency of the global view, knowledge, ontology [Web Ontology Language, 2004], schema, or other equivalent mechanisms for the representation and exchange of the knowledge upon business relations among partners. Usually, such global view is imposed upon business partners by the strongest (or simply the first) partner. Smaller business units have no choice and must obey the mandatory view unless they want to take part in the business. In the case of relatively equal partners, a process of establishing a common view may take a lot of time, as everybody wants to cash in on a situation. Wider is cooperation, lower are the chances to establish a single, consistent common view. Second, to minimize efforts related with future collaboration (with the same or other business partners), an owner of information source usually provides information in a general (universal) and extended (parameterized) way, even if some information/functionality is not needed at the moment. Thus, an interface/functionality of a local system of a business partner is usually quite complex and (sometimes in a large part) unused for a long time. Moreover, an excess of parameters, modes, message types, etc. may substantially complicate an implementation of the information flow at the partners' side. Last but not least – changes in the view (ontology) may force re-implementing large system parts. Note also, that re-engineering is usually not addressed by a community of business partners – an implementation of an ontology/knowledge wrapper (from the global to a local, private view) is a private property of a party. Even if some parties slightly differ, their private information systems are kept disjoined.

The above problem is especially sharp for ad-hoc business collaboration, with dynamically evolving business goals, partners, and information that needs to be exchanged. Ad-hoc collaboration requires a support for step-by-step strengthening of the cooperation among business partners, with the majority of manual work at the beginning, and full automation once after some time the cooperation is stable, well-defined and probably long-lasting. Applying the above-mentioned approach of providing a single generic service for all possible business cases leads to a situation while a cost of providing a useful set of services is too high. Thus, due to the economical reasons, service owners tend to use the services for a long time, in a stable manner. As a result, short-term business relations are usually supported by a manual work of the human staff, with the use of modern telecommunication and information technologies, Web Services included, far below their potential possibilities.

As follows from the above-described limitations, a generic, centralized implementation of a traditional Web Service is inadequate for an efficient implementation of a set a distributed, individual, sometimes temporal and ad-hoc, user-specific activities. A better solution is to implement and use several individual instances of user-defined programs, possibly distributed across the network, working both in the polling (i.e., at user's demand) and pushing (i.e., active-service) modes. Choosing a place, form and time of execution of a given instance should be case- and user-dependent. In a typical case, the instances related with monitoring and alerting of a selected service are to be executed preferably at the server-side, while the software related with the partner-specific information processing (service pipelining, comparison, etc.) should be executed either at the client-side, or at a selected host in the network, if the end-user system/communication line is not powerful enough. However, the above assignment (as well as detailed functionality of each instance) should be adjusted to the specificity of a particular business case and evolving partner's requirements.

Nevertheless, a technology is needed to combine execution of distributed programs, owned by different business parties, orchestrated to achieve case-specific business goals under different restrictions. It seems that the software agent technology may be suitable here. In the next sections, we provide details about using software agents for implementing Virtual Web Services, to be created, managed, and utilized by the end-users.

3. SOFTWARE AGENTS FOR VIRTUAL WEB SERVICES

In our approach, we define software agents in classical way, as presented in [Franklin, Graesser, 1996; Nwana, 1996; Wooldridge, Jennings, 1995; Wooldridge, 2002]. A *software agent* is a program, executed at a given place, characterized by: (1) autonomy – agents process their work independently without the need for human management, (2) communication – agents are able to communicate with one another, as well as with humans, and (3) learning – agents are able to learn as they react with their environment and other agents or humans. As follows from the above definition, an agent may be programmed by its owner, thus allowing unrestricted personalization of behavior of this agent. Agents may be executed in different places, according to owners' needs and possibilities of the end-user hardware [Milojicic, 1999]. Agents may create and use other agents (called *robots*) to perform some remote tasks in a background, for example to continuously monitor a given data source and inform about data changes.

We propose to implement Virtual Web Services using the ACE agent-based framework described in detail in [Rykowski, 2003-A; Rykowski, 2003-B; Rykowski, 2005-A; Rykowski, Juszkiewicz, 2003]. The framework is based on a set of distributed *Agent Servers*, each of them capable of storing and executing software agents. The agents are programmed either in XML, or in Java programming language. The agents may be moved among Agent Servers. Agent Servers may be located in both stationary and mobile devices. In case of stationary equipment, multi-user, multi-agent, mass-usage Agent Servers may be used. In case of mobile devices, characterized by limited hardware and software possibilities and high communication costs, personal, single-agent, light-weight Agent Servers may be used. Depending on hardware and communication restrictions, and current situation, a user has a choice in determining a place of execution of agents. Moreover, as situation changes, the execution place may also change.

Agents communicate among themselves by the use of the XML [XML Protocol, 2000] text-based messages. Agents may also contact local and remote software, both stationary and movable. To this end, specialized agents-brokers are used called *gateways* (for contacting remote software systems) and *drivers* (for contacting local software and hardware). Gateways and drivers are usually prepared by system designers and designed for public, parallel, massive usage. In most cases, in addition to their basic functions, they act as information-cache, programmable wrappers, communication multiplexers, etc.

As already mentioned, agents are created and executed on behalf of their owners. An owner of an agent may be a human, or another agent. An agent is controlled by its owner, i.e., the code is defined by (or at an order) of the owner, and the internal agent variables (agent's state) are defined and accessible for the owner only. Agent's behavior, programmed in its code and variables, is determined by the owner and used for his/her/its individual purposes. Moreover, such behavior may be adjusted to the hardware/software possibilities of the environment an agent currently executes in. For example, several agent's GUI interfaces may be provided and automatically invoked: a textual interface for a mobile phone, limited graphical interface for a PDA, and rich interface for a PC screen [Rykowski, Juszkiewicz, 2003].

Agents differ by state and behavior. Agent's state is represented by current values of agent's internal variables. These values are permanent, i.e., they are conserved while the agent is not activated and restored just before its activation. Agent's behavior is programmed in agent code. To this end, any programming language may be used. However, two basic languages are preferred: Agent Shell Language ASL, an XML dialect for imperative shell-like programming, and Java. The ASL programming improves overall system safety, due to continuous code inspection and impossibility of inserting viruses and other malicious code. However, as the ASL-based agent code is interpreted rather than compiled, agent efficiency is limited. On the contrary, Java agents are compiled and thus efficiently executed, however, by the cost of reduced overall system security. To improve security, Java agents may be defined only by "trusted" users [Rykowski, 2003-A].

3.1 Agent programming mode

The question arises: should we use declarative or imperative way of agent programming? Usually, declarative way of agent programming is preferred, including markup languages and Agent Definition Languages ADL (see Section 4 for more detailed discussion on existing proposals). However, we chose the imperative approach for programming agents, due to the following reasons.

A software agent implementing a virtual service must have a possibility to combine several WS of different input/output data, location, kind, and purpose. Moreover, such service combination has to follow frequent changes in the environment. Thus, it is not possible to use declarative or skeleton-based agents, as they are not flexible enough to deal with different classes of services, different user requirements, and dynamic changes. Instead, imperative programming should be used. Using imperative code, agent's owner may program any behavior of the agent. In the declarative approach, code generation is limited by the declarations or skeletons defined by system designers. Moreover, while collecting information coming from different Web Services, additional data processing is needed – wrapping, formatting, presenting, etc. Such processing must be defined in an imperative programming language, even if all the data sources are declarative-programmed.

The data/knowledge interchange between agents and Web Services is not standardized (as in general it cannot be). There are some proposals for data interchange standards (e.g., SOAP [SOAP, 2003], FIPA [FIPA home page, 2006]), however, these proposals concern physical data transfer, with limited support for semantic data processing. The knowledge representation and sharing standards (e.g., KQML [KQML, 2003]) deal with data semantics. However, these standards do not concern data processing. As a consequence, while collecting data in different formats coming from different sources, additional data treatment is needed. This task must be performed in an imperative programming language, as one cannot foresee declarations for all possible standards, connections, data transfers, etc.

There are some proposals to fill the gap between the declarative and the imperative code, e.g., Rose XDE [Rational Rose XDE, 2002], or embedding declarative code in imperative programs (e.g., embedded SQL for Java/C++), etc. However, it is impossible to foresee expectations of all the users, so one cannot provide a complete set of possible declarations/skeletons/classes, etc.

Taking into account all the above-mentioned system features necessary for unrestricted personalization of Web Services, we decided to allow users program their agents in an imperative way. Of course, the users are allowed to use the declarative mode as well, however, for their individual purposes only.

3.2 System architecture

The main idea of our framework is to use software agents to extend possibilities of existing Web Services by service personalization. We use ACE agents to combine execution of distributed code, owned by different business parties, orchestrated to achieve case-specific business goals under different restrictions. The agents act as user-programmable brokers among distributed Web Services and agents' owners. ACE agents are not used for creating, storing, and providing information – these are the Web Services that are used for. Instead, the agents personalize the way a particular user access the data. The agents are used to create Virtual Web Services (cf. Section 1), where information taken from a particular service is used by another service/agent to provide additional functionality. ACE agents are also used for continuous monitoring of changes of information provided by the Web Services (cf. Section 2.3) and potential alerting once a change expected by a user is detected. What is "expected" and why it is interesting to the owner, is programmed in agent's code by agent's owner. Thus, each agent is (potentially) different, and adjusted to its owner's preferences and requirements. Note that it is not globally possible to replace individual agents by a single universal Web Service, because such service would be too complex and used only in a small part by the particular users.

There are four basic classes of elements of the proposed system: Web Services to be accessed, public Service Agents, Private Agents, and private Virtual Web Services (Fig. 1).

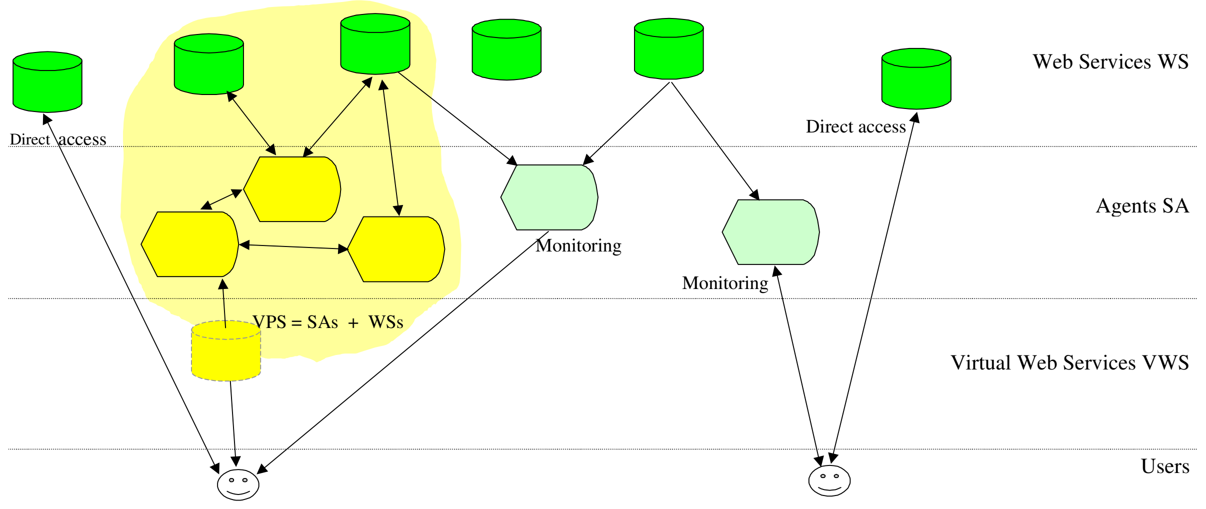

Figure 1. Web Services, agents, Virtual Web Services, and their users

Web Services WSs are treated as external software. They may be accessed and used by agents on behalf of their owners. WSs may be used, but they cannot be changed by users (except for server-side personalization and parameterization, if any provided). Thus, the usage of the WSs is somehow stable and well-defined. We assume that all the WSs are accessed by the agents only. If a WS is to be accessed by a user directly, it is out of the scope of the system, and it may be performed in a usual way, with no personalization utilities. We also assume that the WSs are not linked among themselves. Such links, if any, are of no interest for the users, as they cannot be traced and controlled by end-users. The WSs are programmed using either declarative, or imperative manner – this fact is transparent to their usage and thus it is not reflected in the system in any form. A way of contacting and using a particular WS is defined for this WS, including such information as user accounts and passwords, data semantics, etc. We assume that this information is known in advance for each WS, and that this information may be used by users to program their access to the WSs.

Public *Service Agents* SAs are agents created by trusted users (usually system designers), to be used in a massive manner by many users (either in parallel, or sequentially, depending on the SA). SAs are treated as trustful system elements. Their efficiency is of primary concern. Thus, SAs are programmed in Java, if necessary with some declarative code embedded. Java was chosen for its universality, portability, efficiency, big support for using with Web Services (built-in support and libraries for HTTP, XML, SOAP, KQML, SQL, and other standards), and openness for linking with other software. Standard Java security checking is applied. If needed, some additional security mechanisms may be manually added by programmers, including user account and password checking, token verification, ciphering, etc.

A way of usage of a given SA cannot be changed by an ordinary user. SAs are used in all the situations where there is a need for massive access by many users to given information, providing this information in a standardized form and with optimum effort (from the system point of view). The most frequently used SAs are the following:

- Communication wrappers, allowing communication using given protocol, and hiding data transfer details from end-users (more precisely, other agents owned by end-users). Communication wrappers – drivers and gateways – allow for standardized access to the remote software, including WSs;
- Cache utilities, synchronizing a parallel access of many users to a single sequential Web Service; the information from the service is fetched periodically or at request, and then it is cached and re-sent to other agents if requested;
- Web Service synchronizers and wrappers – a connection of a communication wrapper and cache utility, allowing optimum access to a given WS by many agents;
- Brokers, providing pre-processing of the data, eventually merged from several WSs.

Detailed purpose and type of a given SA is not limited to one of the above proposals. The above functionality may be arbitrarily mixed in a single SA, and several cooperating SAs may be used to provide a single (in this case "virtual") utility for end-users, to be used for preparing individual VWSs.

The *Private Agents* PAs are created and controlled by their owners. There is not a single user, including system designers and administrators, different from the agent's owner, entitled to execute private agent code and access private agent variables. Unless directly ordered by its owner, the agent cannot be accessed by any other agent and service, including all the WSs, SAs, and other PAs.

The main problem related with PAs is safe execution of their code. As discussed in the beginning of Section 3, private agents must be programmed in an imperative, unrestricted manner, and agents' code, from the system point of view, must be treated as potentially dangerous. Thus, the code must be carefully inspected before and during the execution, to detect all the malicious behavior. For many reasons, including pure psychological fear of virus/Trojan attack, the private agent code cannot be compiled. Thus, using any standard Java/C++ approach is not possible, despite all the built-in security checking. Instead, an interpreter should be used to ensure the interpreted code behaves well. Providing agent's code as pure text to be interpreted ensures that the code is free of malicious (or simply errored) code. However, run-time inspection is still a must. Standard interpreted languages, as JavaScript and Basic, are not equipped in continuous security inspection. For example, a never-ending loop (`while (true) do … done`) in JavaScript/Basic may be run without time and space restrictions. Thus, it is easy to block a computer (or at least to consume significant amount of computational power) with quite simple methods, knowingly or not.

Taking into account this disadvantage of the current interpreted languages, we propose a new language called Agent Shell Language ASL, with particular stress put on secure interpretation of the agent code. The ASL language is based on the XML standard [XML, 2005], and its syntax computational power is similar to the widely known shell programming languages [Introduction to C shell, 2005]. Note that we were not able to adapt any existing XML query language [XML Path, 2006; XML Query, 2005], as well as generic XML transformation language [XSL, 2005], as these languages are specialized for node processing and generating a set of XML nodes as a result of processing of queries/other nodes. Instead, we adapted typical shell syntax, adjusting it to the framework of the XML documents (Fig. 2). In the proposed agent-shell programming language, the following shell-language statements may be used: *variable* statement (variable definition), procedure and module definitions (similar to procedure/method and module definitions in most of the imperative programming languages), *if-then-else* and *while-do-done* loops with conditional statements, and procedure *call* with return statement. The language is more expressive than typical shell language, due to unrestricted nesting of XML nodes. Thus, any sub-part of a statement may be expressed statically (as a text node) or dynamically, as a set of nested nodes fixing a text-based result at run-time. As a consequence, in contrast to a typical shell program, where the program structure is fixed, XML-encoded agent's code may be dynamically modified (and self-modified) during its execution.

```
<AGENT name="answer_SMS">
  <CALL name="comm._sendSMS">
    <PARAMETER  name="to"  value="0602123456">
    </PARAMETER>
    <PARAMETER  name="body">
    <VALUE>An SMS arrived!</VALUE>
    </PARAMETER>
  </CALL>
  <RETURN>Thanks for your SMS...</RETURN>
</AGENT>
```

Figure 2. A sample PA encoded in the Agent Shell Language ASL

There are some restrictions on executing agent's code and manipulating agent's variables. First, total agent's execution time is limited to certain number of seconds/minutes (depending on the agent and its owner). Even if the code is willing to execute, the execution is unconditionally stopped once the run-time slot for the agent is over. Thus, never-ending loops and procedures are no more executed forever. Such a mechanism is not used for a typical programming language (including shell and scripting languages) - only program unexpected suspensions are possibly detected and eliminated. As agents are executed autonomously, without direct control of their owners, unconditional time-outs and execution breaks are necessary, unless a small number of agents would consume most of the computational power and system resources.

Second, total number of agent's internal variables is restricted to a certain number of bytes (kB, MB, etc., depending on an agent). As soon as the agent's variables are less than the variable pool area, the agent executes normally. However, once a single variable grows above the variable pool, an execution is terminated, and the new variable value is not set. Thus, agents cannot block the system due to continuous, still growing memory demands.

If any of the above unconditional breaks occur, an agent owner is informed about this fact with a system-generated message. Thus, users are aware of the proper execution of their agents. Moreover, each agent may be equipped with an `on-error` module, automatically executed if such fatal error occurs, and informing agent owner, in an individual manner, about the error and its consequences. In addition, a recovery action may be programmed, which improves a level of agent's autonomy. During the recovery, an agent may be self-reprogrammed to deal with the error during future executions.

Our shell-like agent programming language is not fast (taking into account both the execution time and CPU consumption) and computational-complete, similar to the most of shell and scripting languages. We assume, however, that most of the computations and data processing are performed outside interpreted agent's code. This is a classical situation, similar to a typical operating system, when a shell layer is usually used only for invoking ready-to-use, system-defined executables. Here, previously mentioned SAs and WSs are equivalencies of executables. Complexity of a typical user-defined agent (a shell script) is usually quite low; however, this agent (script) may be responsible for efficient execution of complex software. Note that the executables cannot be changed by ordinary users at the agent/shell level; however, they may be effectively used. Typically, only a small fraction of the code is written by the users, and in consequence, interpreted. Most of the time of agent execution is spent on running Java-based SAs and WSs – compiled and effectively executed system parts. Moreover, from the system point of view, executing a particular agent is not a frequent operation (in comparison with other agents' executions), thus overall system efficiency is high enough. Thus, as it was mentioned in Section 3, even if an efficiency of interpreting particular agents' code is much less important than overall system security, the agents are usually efficient enough.

The last but not least class of system elements, *Virtual Web Services* VWS, are created and used by users in an individual manner, similar to their PAs. As mentioned in Section 2, VWS are complex sets of interconnected agents and services, acting as a single Web Service. A VWS is composed of a set of private agents (belonging to a single user) and links among different SAs and WSs. All the elements in the set are arbitrarily connected, forming a directed graph. A single PA is distinguished to be an entry point to the VWS. The VWS composition and internal links among elements may vary in time (even at run-time). Dynamic evolution of VWS is not necessarily visible by the service owner. Information flow may be different for given executions and it is not necessarily fixed by the VWS composition and links.

In the next sections, basic architectural issues are presented of implementing the idea of Virtual Web Services by the ACE agents.

3.3 Programming levels

Each VWS is composed of private agents of the VWS owner, and some system agents to realize the tasks common for the whole population of users and the Web Services. Thus, in a natural way, each VWS is implemented at two programming layers: the *private layer*, composed of the private agents to be run under individual user control, and the *system layer*, composed of the system agents (gateways to the Web Services included), designed and managed by the administrators. Private agents are used for:

- VWS *composition*, i.e., establishing links among other agents used and external WSs, determining data flow, pipelining, wrapping, formatting, etc.,
- *service personalization*: data formatting and presentation, user preferences, case detection (hardware/software used, communication costs, etc.),
- *advanced tasks*: continuous monitoring with automatic notification of changes, comparison of data coming from different sources and at different time, user-dependent wrapping and formatting, caching, etc.

Private agents may be activated either at request, or in an asynchronous manner. In the first case, this is the VWS owner who initiates the service activation, asking for some data. Once the execution of all service activities is over, a service response is generated and sent to the owner. The VWS becomes inactive until the next request. In the second case, this is the VWS who initiates the activation, usually after detecting an information change in one of the observed Web Services. Once a change is detected, and this change is of interest to the VWS owner, an appropriate message is generated to the owner. Thus, agents of the private layer may be used for continuous monitoring of changes of information provided by the services.

A structure of the private layer of a given VWS is not determined in advance. Moreover, it may evolve in time, as situation changes, and user requirements and preferences evolve. Only basic mechanisms are provided a priori: agent mobility, agent execution environment, XML messaging, etc. The PAs composing the private layer have no fixed structure, they are individually programmed by their owners. However, some global, additional mechanisms may be used as well - ciphering keys, user accounts, passwords, GUI specialized for given hardware/software, etc.

System agents composing the *public layer* cannot be modified by the VWS owner; however, they may be accessed in the parameterized manner by the agents of the private layer (cf. Section 2). Basic tasks of the public layer are the following:

- information storing, extracting, and processing, possibly on the mass scale and in the parallel manner, with access synchronization, authorization, etc.,
- massive and repetitive computing,
- standardization of data access methods and service interfaces, for several different Web Services of different functionality and business purpose,
- high-level wrapping (in a mass rather than personalized manner),

- data caching, which is especially useful for frequently accessed Web Services propagating business information to many agents in parallel,
- mass synchronization of parallel access to common data, etc.

On the contrary to the private layer, functionality of the public layer is determined in advance. The system agents, gateways to the Web Services included, are prepared by system designers and outside programmers, and thus they are known in advance for the VWS owner. The VWS owner may use the pool of available agents from the public layer, having a choice. However, any system agent used cannot be changed, extended, modified, etc. Usually, the available system agents are treated as a library of independent elements, to be individually used for VWS composition.

Basic advantages of splitting a VWS into the private and public layers are the following.

- The private layer is under exclusive control of the service owner. This layer is a personalized broker between the Web Services (via system agents) and the owner. Thus, almost any Web Service may be personalized to the maximum extent, without a need for any modification/personalization at the server-side.
- The approach is open to newly incoming standards and services – the service updates may be quickly and effectively incorporated.
- Accessing the services is performed under the system control, in the public layer. No direct access (i.e., without system agent's mediation) is possible by a private agent, thus some additional activities may be undertaken to improve overall system safety (access logs, passwords, tokens, etc.).
- The owner may introduce, in the private layer, his/her individual security mechanisms, to protect him/herself against malicious services.
- The private layer may be used for intelligent, personalized filtering of the information gathered from the external world.
- The computation overhead, in comparison to the direct access to Web Services, is quite small, while the obtained personalization level – very high.
- The agents, both from the private and public layer, may be executed at different hosts (including owner's mobile devices), thus using all the advantages of a distributed computing environment (parallel execution, efficiency, low transmission costs, load balance, etc.).
- Several independent of even competitive Web Services may be integrated, from a user point of view, into a single Virtual Web Service.

3.4 Support for modern telecommunication channels

As the ACE agents have to exchange information with their owners in an efficient manner, despite some limitations caused by current owner's localization, hardware, and connection link, it is quite important to provide direct access to the agents via modern telecommunication facilities, mainly mobile phones [Talking point, 2002]. Thus, specialized SAs are provided being gateways to selected public telecommunication systems, to enable contacts with humans via communication channels of different type and purpose [eMobile, 2005]. In general, two basic types of communication channels are available: textual and Web-based. A *textual channel* is able to exchange flat (unformatted) text messages, usually among humans and agents. Physically, textual channels may use such media as an e-mail SMPT/POP3 connection, SMS (Short Message System)/MMS connection with a telecommunication network, a voice gateway, etc. Once sent by a textual message, an ACE agent acts as a chatterbot [Jurafsky, 2000], analyzing the message via keyword extraction and analysis [Zillman, 2003]. The semi-natural access to an agent in a chatterbot manner is especially useful for non-advanced users, and for users temporary handicapped due to limited hardware possibilities and communication costs. For example, an SMS message may be used to check the most important information during a journey, once a stationary PC is further used to get the complex information while the user is back home [Rykowski, 2005-E].

Web-based channels are used to access an agent via a WWW/WAP page, and from specialized ACE applications. These channels use personal, semi-automatic formatting of both contexts and presentation of the data to be sent. To this goal, XSL-T technology was adapted with XSL transformations defined in a personal manner and stored in private agent variables [Rykowski, Juszkiewicz, 2003]. In a case of a conversation with a human, automatic detection of end-user device may be used, thus restricting the communication. For example, a small textual message is sent to a mobile phone using WAP connection, similar message with the same contents however some additional formatting is sent to a PDA device, and full text&graphic message is sent to a stationary PC.

Number and types of the telecommunication gateways used (including some specific parameters, as a phone number for an SMS center, an address for a SMTP/POP3 server, etc.) is local-administrator dependent. Note once again that these gateways are implemented as system agents, thus one may easily extend a given Agent Server by some specific communication means.

Once ACE framework is used in a mobile-telecom environment, it is quite usual to use telephone numbers as global identifiers of agent owners. Such numbers are unique across the whole telecommunication network; moreover, these numbers are independent of current end-user device, communication links, a geographical place of access, etc. Note that the phone number of a telephone owner is very difficult to falsify. As a telecom network and all its connections with other networks are owned and controlled by a telecom operator, there is no user-accessible mechanism to control and change telephone numbers. As a consequence, a telephone number, which is transferred by the very beginning of each connection, may be used as strong authentication mechanism[1]. It is up to a user to apply in addition standard names/passwords and more advanced authentication mechanisms, as for example specialized SIM cards, PINs, tokenizers, etc. However, a usage of these mechanisms should be pointed out in the code of user's private agents.

3.5 An example of a Virtual Web Service

An example of a VWS is presented in Figure 3. The VWS is composed of two private agents, more or less five (depending on the just-served requests) system agents, and two Web Services. Agent PA_1 (Figure 4) is an entry point to the service. There are three basic ways to contact this agent, so called contexts: SMS context, e-mail context (both textual communication channels), and HTTP context (menu-based channel). To distinguish the code that is responsible for serving the above contexts, contextual interpretation is used. To this goal, the "context" attribute of each ASL tag marks the visibility of the marked tag at the run-time in a given context. The default context is "any", i.e., if not explicitly mentioned, the tag appears in any context and is always interpreted at run-time. Contextual interpretation increases the clarity level of the ASL code, thus it may be effectively used to program ASL-encoded private agents [Rykowski, 2006-E].

Agent PA_1 invokes several system agents: SA_2, the main gateway to the Web services, SA_1, the natural-language/chatterbot parser and keyword detector, and SA_3, the menu-based preprocessor converting menu-based invocations to the standardized form.

On the contrary to the user-generated requests to the PA_1 agent, an asynchronous contact is invoked by the VWS itself, by sending an SMS/MMS/e-mail message. To this end, PA_2 agent is used for monitoring given external Web service and informing service owner (via selected personal communication channel) about detected information changes. Agent PA_2 uses some additional system agents SA_m and SA_n, in order to provide long-lasting comparison for selected information changes (here: a price list) from a Web Service and alerting the agent owner once any change is detected. The agent is executed according to the "cron" context (automatic code execution at certain moment in time), periodically every 15 minutes excluding certain days of a week (here: Sundays) and night hours.

[1] We pass over a possibility of a mobile phone theft, as this is up to the user to block a device/SIM card once it is stolen.

Note that the ASL code, as well as the internal organization of a Virtual Web Service, strongly depends not only on the agent owner and his/her individual programming skills, but also on the availability of certain system agents, detailed way of parameterization of the invocations of these agents, etc. Thus, the above described service should be treated as a reference only, to provide some overview on the personalization level offered by the ACE agents. We think that due to the contextual interpretation and several comments included, the code is self-described and need no other explanation. However, the private agents presented here are only a small example of the possibilities of the ACE framework.

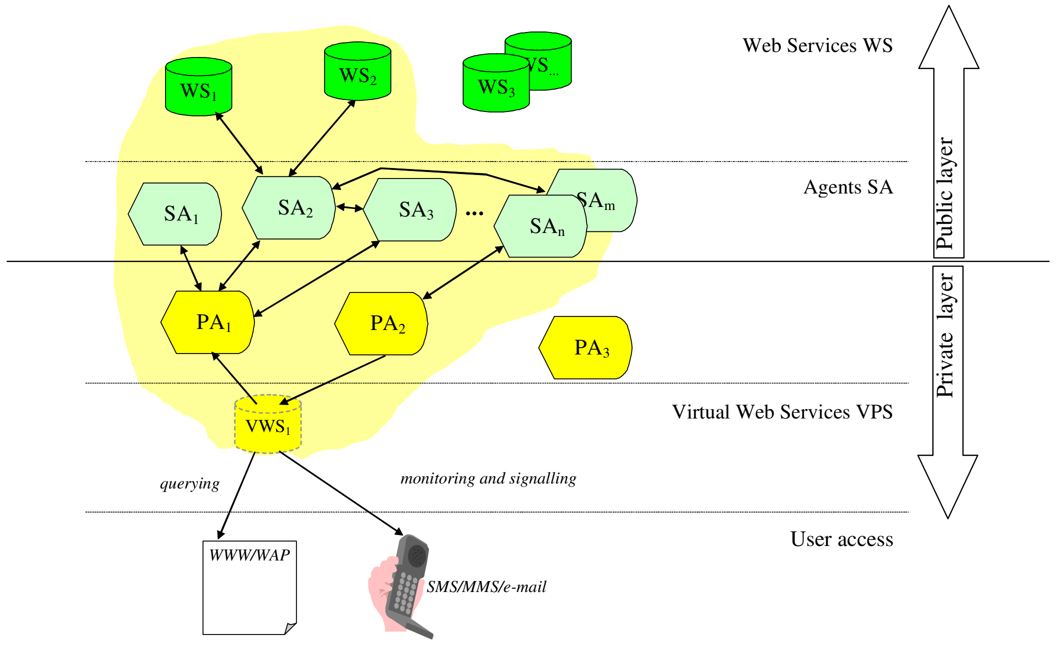

Figure 3. Public and private layers of a sample Virtual Web Service

```
<AGENT name="PA1" context="sms, mail, http">
  <VARIABLE name="servicesToRun" priority="1" context="sms, mail">
    <VALUE> lotto weather calculator exchange stocks</VALUE>
  </VARIABLE>
  <VARIABLE name="response" priority="10">
    <VALUE context="sms, mail">
      <CALL name="SA2"> <!— call ‘Textual Service’ -->
        <PARAMETER name="message_content" value="{message_content}"></PARAMETER>
        <PARAMETER name="service_name">
          <CALL name="SA1"> <!— call ‘Guess Service Name by Keywords’ -->
            <PARAMETER name="input" value="{servicestorun}"></PARAMETER>
            <PARAMETER name="{message_content}"></PARAMETER>
          </CALL>
        </PARAMETER>
      </CALL>
    </VALUE>
    <VALUE context="http">
      <CALL name="SA3"> <!— call ‘Menu-Based Service’ -->
        <PARAMETER name="service_parameters" value="{HTTPparameters}"></PARAMETER>
      </CALL>
    </VALUE>
   </VARIABLE>
   <RETURN priority="20">{response}</RETURN>
</AGENT>

<AGENT name="PA2"
       context="cron"
      cron="every 15 min minimum 8:00 maximum 23:00 +weekday -sunday">
  <VARIABLE name="lastserviceresult">
    <VALUE>
      <CALL name="SAn"> <!— call 'Compare And Alert Service' -->
        <PARAMETER name="lastvalue" value="lastserviceresult">
        <PARAMETER name="currentvalue">
          <VALUE>
            <CALL "SAm"> <!— call 'External Web Service' -->
              <PARAMETER name="service" value="WS2"></PARAMETER>
              <PARAMETER name="input" value="ask_for_current_price_list"></PARAMETER>
            </CALL>
          </VALUE>
        <PARAMETER name="service_prefix" value="Price change:"></PARAMETER>
        <PARAMETER name="alert_type" value="sms"></PARAMETER>
        <PARAMETER name="sms_number" value="0999123456"></PARAMETER>
     </CALL>
   <VALUE>
  </VARIABLE>
</AGENT>
```

Figure 4. Sample code of private agents of a Virtual Web Service

Regardless the technical details that are related with the code of the private and system agents, assume that WS_1 and WS_2 are Web services owned by the producers and resellers of car equipment. These producers are partially competitors, i.e., some equipment is offered by two of them, while some – only by one of them. The prices of the goods offered as well as the accessibility of these goods may vary in time. The VWS service owner – a car-assembly company – is interested in obtaining the just-in-time (JIT) delivery of the goods with a minimum price.

The Web Services of the producers are not inter-connected, because they are competitors on the same market. Thus, no public Web Service is available to compare the offers of these two companies. Instead, VWS should be used in the following manner. Once a new JIT delivery is needed, the PA_1 agent (users' entry point to the VWS) is requested. The agent is responsible for formulating the order form. This order is sent to the SA_2 agent to choose the best (at the moment) supplier. The competitive Web Services are contacted and the offers of the corresponding suppliers are compared. To provide the comparison of prices and the detailed checking of delivery limitations (quantity, timings, etc.), some specialized system agents are used. Finally, better offer is chosen and the appropriate Web Service is contacted to initiate the delivery. The status message is sent back to the PA_1 agent, and in turn, to the VWS owner.

Once current conditions for the delivery are not met by the suppliers, the VWS owner would be probably interested in obtaining information immediately these conditions are fulfilled by at least one of the competitive suppliers. To this end, PA_2 agent is used that is instructed to periodically poll for the changes of the information provided by the competitive Web Services, via the SA_2 gateway. Once a new offer is detected that is satisfactory for the VWS owner, appropriate message is sent to the VWS owner. The owner is then able to start a new delivery in the above-described manner.

Note that the above monitoring and alerting is a personalized process – waiting for the given service offer to be changed to satisfy VWS owner's private criteria. The notification process is situation- and user-dependent as well. The final business-action conditions may be different for different users. Thus, it is not reasonable to provide comparison and monitoring functionality at the public level. Instead, such personalized tasks should be programmed individually for each end-user, in the code of the private agents.

4. RELATED WORKS

To our best knowledge, the idea of Virtual Web Services is a new idea, and there are no proposals dealing with software agents used for user-defined personalization and integration of Web Services. Thus, we cannot globally compare our approach with any other similar approach. However, we may provide a comparison of a functionality of our personalization method and other personalization methods proposed for Web Services, as well as we may compare some aspects of using software agents for programming Web Services: using imperative vs. declarative programming for defining agents' behavior, using XML for programming agents (both code and state), using software agents for defining Web Services, combining Web Services into a single service, and client-side personalization of services.

Between two basic approaches to build software agents – declarative and imperative, the declarative approach is more popular. The declarative techniques are based either on agent patterns, definitions, skeletons, descriptions, or a specialized Agent Definition Languages ADLs.

A pattern/skeleton is a definition/prototype of a sample, specialized agent. Such definition (class) may be instantiated with certain parameters, making it possible to execute in an individual manner. As patterns/skeletons are prepared by system designers rather than ordinary users, their code may be treated as trusted and run without careful inspection. Code personalization depends on parameters defined by individuals, so the behavior of agents may be adjusted to users' demands and current situation. Such approach provides good results in closed environments, with well defined functionality. An example application area is e-commerce, with agents being sellers and buyers; e-manufacturing, with agents specialized for a machine tool; distance learning, with agents controlling students and helping teachers, etc. Using pre-defined patters/skeletons is however restricted in an open, widely distributed environment, for at least three reasons. First, "artificially" (from a user point of view) created agents are "black boxes" for their users. A user may create and use agent of a given type, however, he/she cannot fully program and control its behavior. Second, as instantiation parameters are the only way of agent individualization, agents behave in a similar manner. Users cannot use personalization to a great extent. Third, possible number of agent patterns/skeletons must be kept small (usually a few), in order to effectively manage them by both the system and the individuals.

The AMASE prototype [AMASE, 2002] is an example of the "skeleton-based" approach. AMASE agents are built from basic, stable definitions. The main stress is put on location awareness and similar look-and-feel regardless the hardware used. The agent code, provided by administrative staff, cannot be changed to a great extent. As a consequence, AMASE system may be used in closed environments, where security is of primary concern. This system, however, is not well suited for an open environment and user-defined agent behavior.

The WhereWho system is an example of a distributed search engine, based on agent technology and agent-skeletons. This system is primarily a reference source or catalog that binds information about resources, point of interests, people or other features to precise geo-locations. Agents issue WhereWho queries, and then perform customization, filtering and direct agent-to-agent transactions with the entities whose contact information is provided by the query results [Youll, Krikorian, 2000]. The system uses specialized agents with system-defined behavior and is an example of a closed agent environment.

To improve flexibility of an agent system in comparison to the above pattern-based approaches, Agent Definition Languages are used. An ADL is usually specialized for given application areas, agent types, environments, etc. There are some basic classes of ADLs: based on relations of type message-action (message-driven), based on logical programming (like Shoham's Agent-0 language [Shoham, 1993]), based on markup languages (XML, RDF and similar standards), and based on target application area (task- or problem-oriented, e.g., for e-commerce, e-cooperation, e-manufacturing, etc.).

Due to growing popularity of the XML standard, some proposals are proposed based on XML, RDF [RDF, 2006] and similar markup languages. One of the most advanced is DAML (DARPA Agent Markup Language). DAML is a technology with intelligence built into the language through the behaviors of agents, programs that can dynamically identify and comprehend sources of information, and interact with other agents in an autonomous manner. DAML agents can be embedded in code and maintain awareness of their environment, are user-directed, but have the capacity to behave autonomously. DAML includes a type of query language with a specialized ability to find and process relevant information [DARPA, 2006]. DAML proposal is going towards Web Services applications and "intelligent" agents [DAML, 2006] where one says "what" to do, but not necessarily "how" to do. The imperative programming is not used; instead, markup-based queries are the computational and transactional base. However, using the query language limits an application scope of the

DAML proposal and requires using classical programming languages in parallel, at least for data formatting, communication, etc.

Agentool [AgenTool, 2002] is a system in which agents are automatically built based on formal description. This is another example of closed environment based on the "skeleton" approach, with "skeleton" parts equivalent to parts of the formal description. The FIPA (Foundation For Intelligent Physical Agents) [FIPA home page, 2006] document on personal agents [FIPA Personal Assistant, 2006] is a proposal (in experimental state) for a standard for the schema of interactions between an agent and its user (and other agents). The main goal of this proposal is to define an architecture and functionality for a personal agent-assistant, with particular stress put on communication scenarios and agent ontology. The proposal mentions neither multi-channel communication and device-adaptation problems nor a need for used-defined agent's code.

Defining agents at higher levels of abstraction, in comparison with classical programming languages, leads to improved system security, however, also limited functionality. The executable code is generated automatically by the system, so a danger of malicious agent behavior is substantially reduced. The potentially dangerous user-defined code may be detected before the agent is about to execute, and immediately eliminated. However, the functionality of a user agent is restricted by the power of the ADL used. Usually, the ADLs are specialized for given architectures (e.g., message parsing, action- and problem-driven approaches, etc.), given applications (e-commerce, communication, etc.), given environments, build-in agent "intelligence", fixed agent schema, etc. Thus, there is no one single ADL fulfilling all the required functionalities, and the choice of the most adequate ADL depends on application area.

In the above-mentioned proposals, XML standard is widely used for defining agents and message passing. However, there are no attempts to use XML in an imperative manner, as a programming language for defining agents' behavior. As it was proven in our approach, the XML-based imperative declaration of agents is quite effective and easy to implement. Due to wide spectrum of XML parsers and analyzers, it was possible to link code interpretation with some additional, no standard mechanisms, as continuous code inspection, advanced syntax verification, run-time behavior analysis, etc. Moreover, programming code and state of XML-based agents is uniformed, including XML-based messaging among agents and the environment.

In Active XML (AXML) approach, the AXML documents contain embedded calls to Web Services. Such documents are enriched by the results of invocations of the service calls they contain [Abiteboul, 2007]. The AXML model also defines AXML services, which are Web services that exchange AXML documents. By the exchange of distributed AXML documents, several systems dynamically collaborate to perform specific data management tasks. Our approach differs from Active XML in several ways. First, in AXML, the information is stored in the XML format only, while in our approach, we are able to process any information stored in any (web-accessible) file in any format on any Web server. Second, the basic AXML programming method is related with XML-processing language – an assumption is made that quasi-XML invocation is replaced by XML-compliant result. Thus, it is quite difficult to express some programming tasks such as data formatting and presentation – these must be programmed elsewhere. Third, Active XML approach covers no self-adaptation to the limitations of communication means and end-user devices. Fourth, no persistency and self-activity (i.e., monitoring information changes and alerting users) is discussed, while in our approach, persistent code is a key for user-defined, server- and network-side activities. Fifth, the Active XML is rather not related with user-specific solutions, as the XML schema (i.e., real meaning of the XML tags) is common for all the AXML documents. In our approach, each user is responsible for his/her own set of ACE agents, thus individualizing system behavior to the great extent.

Another approach to define software agents consists in the use of a classical, imperative programming language. However, as agents are theoretically used in an open environment, the used programming technique should be portable and universal. Java is a popular example of such open programming environment, and thus it is widely used for programming agents' code. ActiveX is another similar solution, however restricted to the Microsoft's world of PCs and Windows.

Java-based agents are very popular, as they are easy to develop, and many tools and techniques exist extending the basic standard. Moreover, Java agents need no special runtime environment to remotely execute agents, because Java Virtual Machines exist on most of the network hosts. One of the first Java-agent-based systems was proposed by IBM and named “aglets” [Aglets Documentation, 2003]. An “aglet” was a closed, stateful piece of Java code, which may move to given destination and continue execution there. AMASE [AMASE, 2002] is an agent-based system with agents working in different mobile environments. Voyager [Voyager home page, 2003] is another example of usage of Java-based agents. This is a commercial system, being a base for many agent applications. As for aglets, Voyager agents are movable, stateful pieces of Java code. However, there are no restrictions in stopping and resuming the execution, and Voyager agents may be quite complex entities. Due to its universality and rich functionality, the Voyager system was successfully adapted for many e-commerce and e-business applications. Recently proposed Cougaar [Cougaar, 2006] uses agents composed by users with agent parts chosen from a set of well-defined, stable plugins, encoded in Java by the system designers.

The most important problem of the Java-based agents is to achieve reasonable level of global system safety. Executing external Java code means, from a local system point of view, executing alien code, unknown and potentially dangerous. Even if the level of security provided by Java is considered as to be high, and several additional mechanisms are used – ciphering, digital signatures, anti-virus checkers – one cannot be sure the just executed code behaves well. Maybe in several cases this is more psychological than real menace; however, such approach is usually used in a closed, mutually trusted environment. As a consequence, Java-based agents with unrestricted, used-defined code are used quite rarely, and only in limited, closed environments. In most of the cases, the users are not allowed to prepare the Java code. Instead, system-defined code is used as a set of “black boxes” (e.g., Cougaar plugins).

Our previous work related with personalized access to the Internet information sources and Web Services [Rykowski, 2003-A; Rykowski, Cellary, 2004] was concentrated on personalization aspects related with end-user devices and communication means, mainly a user-defined (user-chosen) data formatting and presentation. This paper broadens this early work by providing final system architecture and pointing out several important implementation issues. In addition, we take into consideration some new application areas, putting stress not only on the final stage of the process of gathering and presenting the information to the users, i.e., data presentation at the end-user device, but also to the other stages related with searching for, collecting, processing and finally returning the results to the users, including server-side monitoring and alerting by the use of user-defined agents.

5. CONSLUSIONS

In this paper we propose to apply user-defined software agents to build Virtual Web Services, being combinations of several virtual and real services, accessed by the user as it would be a single service. Virtual services are created for and by particular users, allowing rich personalization of service functionality, data format and presentation, adjusting to different hardware and software environments. Our approach fills the gap of non-availability of user-defined, personalized mechanisms for service orchestration and comparison, allowing combining different, even competitive services in an automated and agent-supported manner.

The main goal of Virtual Web Services is twofold: increasing a level of personalization of individual Web Services, and allowing a combination of several independent Web Services into a single, consistent service.

The Virtual Web Services make it possible to define individual brokerage for particular Web Services, and thus to personalize the behavior of the services to the maximum extent. In today’s applications, the level of personalization of Web Services is restricted, due to potential service complexity and possible decreasing of the overall system security. The Virtual Web Services fill this gap, because they allow using user-defined, but safe code in different places of the network.

The Virtual Web Services make it possible to combine several independent, potentially competing Web Services into a single, consistent service, personalized for a given user. In current systems, a combination of services is not possible at user level. Instead, in order to use several services to fulfill a given task, the user must fetch a lot of data from the services and assure adequate data processing at client-side. Such a process is time-, and memory-consuming, and requires extensive use of communication links. Thus, for some limited hardware/software environments (like mobile systems for example) it is not possible or economically justified to combine the services at client-side.

The system as a whole is safe, even if a remote execution of user defined code is. The user code is interpreted and carefully inspected at run-time against malicious behavior. The user code may call for execution some trusted system code (public agents) in order to effectively perform some repetitive and global tasks. Thus, overall system efficiency is high enough.

The level of personalization of Web Services is high, and it may be individualized for each user separately. The personalization includes not only single services, but also an arbitrary chosen combination of different, independent services. The personalization also includes an automatic adjustment to current situation, end-user hardware/software used, user rights, etc.

The system is open for new services, communication standards, users, etc. Due to brokerage of public agents, the new services and protocols may be added in a transparent (for an ordinary user) way.

The system is open for new services, communication standards, users, etc. Due to the brokerage of public agents, the new services and protocols may be added in an invisible (for an ordinary user) way.

To our best knowledge, the idea of Virtual Web Services is original, and there are no proposals dealing with software agents used for user-defined personalization and integration of Web Services.

Some work is under run to test Virtual Web Services in a real environment, with commercial services and different users. Previous applications of the Agent Computing Environment – individualized bank access and telecom information services – proved the usefulness of the idea of using user-defined software agents for mass personalization purposes. Our recent proposals cover also e-office support [Rykowski, 2005-D] and individual, targeted marketing in supermarkets, as well as ad-hoc virtual enterprises [Rykowski, 2006-B].

REFERENCES

1. Abiteboul, S., Benjelloun, O., Milo, T. (2007), The Active XML project: an overview, PDF file from ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-331.pdf
2. ACT-NET Consortium (1996). The Active Database Management System Manifesto: A Rulebase of ADBMS Features. ACM Sigmod Record 25-3, pp. 40-49
3. Aglets Documentation and Source Code page (2003), from http://www.trl.ibm.com/aglets/
4. AgenTool home page (2002), from http://en.afit.af.mil/ai/agentool.htm
5. AMASE: Agent-based Mobile Access to Information Services (2002), ACTS project homepage, from http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch3/amase.htm
6. Bettstetter C., Vögel H-J., Eberspächer J. (2002), Technische Universität München (TUM), GSM Phase 2+ General Packet Radio Service GPRS: Architecture, Protocols, and Air Interface, from http://www.comsoc.org/pubs/surveys/3q99issue/bettstetter.html
7. Bonett, M. (2001), Personalization of Web Services: Opportunities and Challenges, Ariadne Issue 28, from http://www.ariadne.ac.uk /issue28/personalization/intro.html
8. BPEL Business Process Execution Language for Web Services version 1.1 (2006), from http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
9. Ceri S., Fraternali P. (1997), Designing Database Applications with Objects and Rules. Addison-Wesley, ISBN 0-201-40369-2
10. Cougaar Architecture Document (2006), BBN Technologies docs, from http://cougaar.org/docman/view.php/17/170/CAD_11_4.pdf
11. DAML Semantic Web Services (2006). From http://www.daml.org/services.
12. DARPA Agent Markup Language Homepage (2006), from http://www.daml.org/
13. Deakin N. (2004), XUL Tutorial, from http://www.xulplanet.com/tutorials/xultu/
14. eMobile SMS Software Solutions (2005), from http://www.emobile.com.sg

15. FIPA home page (2006), from http://www.fipa.org/
16. FIPA Personal Assistant Specification (2006), from http://www.fipa.org/specs/fipa00083/XC00083B.html
17. Franklin S., Graesser A. (1996). Is it an Agent, or just a Program?
A Taxonomy for Autonomous Agents. Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag
18. Introduction to C shell (2005), from http://docs.freebsd.org/44doc/usd/04.csh/paper.html
19. Jurafsky D., Martin J.H. (2000). Speech and Language Processing:
An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice-Hall
20. IMesh Toolkit – Personalization (2003), from http://www.imesh.org /toolkit/work/components/personalization/
21. KQML Agent-Communication Language (2003). The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, from http://www.cs.umbc.edu/kqml/kqmlspec/spec.html
22. Lombardi, V. (2002), Designing for Web Services, from http://www.newarchitectmag.com/documents/s=2452/new1015627350101/index.html
23. Microsoft's Web Services Developer Center home page (2005), from http://msdn.microsoft.com/webservices/
24. Milojicic D. (Ed.). (1999). Trend Wars – mobile agent applications. IEEE Concurrency, 7-8, 80-90
25. Nwana, H. (1996). Software Agents: an overview. Knowledge Eng. Review 11/3, 205-244
26. Oracle Triggers (2006), from http://www-rohan.sdsu.edu/doc/oracle/server803/A54643_01/ch15.htm
27. Paton N. (1999), (Ed.), Active Rules in Database Systems, Springer, ISBN 0-387-98529-8
28. Pinsdorf U., Peters J., Hoffmann M., Gupta P. (2002), Context-Aware Services based on Secure Mobile Agents, Proc. of 10th Int. Conference SoftCOM 2002, pages 366-370, from http://www.semoa.org/docs/papers/pinsdorf2002c.pdf
29. Rational Rose XDE Developer home page (2002), from http://www-306.ibm.com/software/awdtools/developer/rosexde/
30. Resource Description Framework (RDF) home page (2006), from http://www.w3.org/RDF/
31. Rykowski, J. (2005-A). ACE Agents – Mass Personalized Software Assistance. Lecture Notes in Artificial Intelligence, 3690, pp. 587-591
32. Rykowski J. (2003-A). Agent Technology for Secure Personalized Web Services. In Proceedings of the 24th International Scientific School ISAT 2003, Szklarska Poręba (Poland); September 2003; pp. 185-193
33. Rykowski, J. (2003-B). Databases as repositories for software agents. In: B. Thalheim and G. Fiedler (Eds.), Emerging Database Research in East Europe. Proceedings of the pre-conference Workshop joined with the 29th VLDB Conference, Berlin, Germany; 2003, pp 117-123
34. Rykowski, J. (2006-A), Electronic Activity Interchange EAI – a new way of B2B cooperation, in IFIP International Federation for Information Processing, Volume 226, Project E-Society: Building Bricks, eds. Suomi, R., Cabral, R., Hampe, J., Felix, Heikilla, A., Jarvelainen, J., Koskivaara, E., Boston: Springer, pp. 266-278
35. Rykowski, J. (2006-B), Information management by software agents in ad-hoc virtual enterprises, in "Knowledge and Technology Management in Virtual Organizations: Issues, Trends, Opportunities and Solutions", eds. G.D. Putnik, M. M. Cucha, Idea Publishing Group, London, pp. 306-333
36. Rykowski J. (2006-C). Management of information changes by the use of software agents. Cybernetics and Systems, vol. 37, no 2-3. Taylor & Francis Publishing, Philadelphia (US), ISSN 0196-9722, pp. 229-260
37. Rykowski, J. (2005-D), Using software agents to personalize access to e-Offices, 5th IFIP Conference on e-Commerce, e-Business, and e-Government I3E'2005, Poznan, Poland, pp. 95-110
38. Rykowski, J. (2005-E). Using software agents to personalize natural-language access to Internet services in a chatterbot manner. In Proceedings of the 2nd International Conference Language And Technology L&T'05, Poznan, Poland, April 2005
39. Rykowski, J. (2006-D), Who should take care of the personalization?, in IFIP International Federation for Information Processing, Volume 226, Project E-Society: Building Bricks, eds. Suomi, R., Cabral, R., Hampe, J., Felix, Heikilla, A., Jarvelainen, J., Koskivaara, E., Boston: Springer, pp. 176-188
40. Rykowski, J. (2006-E), The Personalized Access to Heterogeneous Distributed Information Sources by Means of Software Agents, Habilitation Thesis No 26, AE Press, Poznań (Poland), ISBN 978-83-7417-206-6
41. Rykowski J., Cellary W. (2004). Virtual Web Services - Application of Software Agents to Personalization of Web Services. In Proceedings of the 6th International Conference on Electronic Commerce ICEC 2004, Delft (The Netherlands), 2004 ; ACM Publishers; pp. 409-418
42. Rykowski, J., Juszkiewicz, A. (2003). Personalization of Information Delivery by the Use of Agents. In Proceedings of the IADIS Int. Conference WWW/Internet 2003, Algarve, Portugal, 2003, pp. 1056-1059
43. Shoham, Y. (1993), Agent-oriented programming, Artificial Intelligence, 60(1):51-92
44. SOAP Version 1.2 (2003). Part 1: Messaging Framework, W3C Recommendation, from http://www.w3.org/TR/soap12-part1/

45. SOA Services Oriented Architecture and Web Services (2004), from http://www-306.ibm.com/software/solutions/webservices /overview.html IBM
46. Talking point software prototype (2002), from http://www.microjava.com/presentations/talkingpoint.pdf
47. UDDI Business Registry Version 2 home page (2005), from https://uddi.ibm.com/ubr/registry.html.
48. Voyager home page (2003), from http://www.recursionsw.com/products/voyager/voyager.asp
49. Web Ontology Language OWL (2004), from http://www.w3.org/2004/OWL/
50. Web Services Activity home page (2002), from http://www.w3.org/2002/ws/
51. Wooldridge, M., Jennings, N.R. (1995-B). Intelligent agents: theory and practice. Knowledge Engineering Review 10/2, 115-152
52. Wooldridge M., Jennings N. (Ed.). (1995-A), Agent Theories, Architectures, and Languages: a Survey, in Intelligent Agents, Berlin, Springer-Verlag
53. Wooldridge, M. (2002), Introduction to MultiAgent Systems, ISBN 978-0-471-49691-5, Wiley &Sons
54. WS-CDL Web Services Choreography Description Language Version 1.0 (2006), W3C Candidate Recommendation 9 November 2005, from http://www.w3.org/TR/ws-cdl-10/
55. WSDL Web Services Description Language v. 1.1 (2001), W3C documentation, from http://www.w3.org/TR/wsdl
56. WSFL Web Services Flow Language v. 1.0, (2005), from http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
57. WSXL Web Service Experience Language Version 2 (2002), from http://www.ibm.com/developerworks /webservices/library/ws-wsxl/
58. XLANG homepage (2005), from http://www.ebpml.org/xlang.htm
59. XML Extensible Markup Language home page (2005), from http://www.w3.org/XML
60. XML Protocol Working Group home page (2000), from http://www.w3.org/2000/xp/Group/
61. XML Path Language (XPath) Version 1.0 (2006), from http://www.w3.org/TR/xpath
62. XML Query (XQuery) home page (2005), from http://www.w3.org/XML/Query
63. XSL Extensible Stylesheet Language Family home page (2005), from http://www.w3.org/Style/XSL/
64. Youll J., Krikorian R. (2000), WhereWho Server: An interactive location service for software agents and intelligent systems, the Second International Symposium on Handheld and Ubiquitous Computing, Bristol (UK)
65. Zillman M.P. (2003). Chatterbot resources and sites, from http://chatterbots.blogspot.com/