Architecture and Modeling of Service-Oriented Systems

Jaroslav Král and Michal Žemlička

Charles University, Prague, Faculty of Mathematics and Physics,
Department of Software Engineering, Malostranské nám. 25, 118 00 Praha 1, CZ*
{kral,zemlicka}@ksi.ms.mff.cuni.cz

Abstract. Service-oriented software systems (SOSS) are becoming the leading paradigm of contemporary software engineering. SOSS are virtual peer-to-peer systems of autonomous software components (services) behaving like the service in mass services systems of real world. SOSS used in *e*-commerce (alliances) have properties different from the properties of many systems (confederations) in which (almost) all their services are known. Examples of confederations are *e*-government or information systems of decentralized enterprises, etc. Confederations admit various implementations being difficult to describe by UML diagrams. The reason is that UML is based on object-oriented philosophy appearing to be different from the service-oriented one. We show that the models of the logical architecture of confederations must combine and generalize some features of data-flow diagrams,the diagrams of Petri nets, and include some features of UML diagrams. The complexity of the model is caused by the fact that different parts of the confederations may use different interfaces and different communication types like database sharing, message passing, and method invocation. Such a complexity probably limits the applicability of the philosophy of model driven architecture for service oriented systems.

1 Introduction

Current software systems development methodology is changing from logical monoliths (possibly physically distributed) to the systems having the character of networks of autonomous components (services) integrated as black boxes (their interfaces only are known). We shall call such systems *service-oriented software systems* (SOSS). SOSS must be implemented as a structure containing the services and an infrastructure – middleware – allowing cooperation and communication of the components.

The properties of SOSS depend on properties of the languages accepted by the interfaces of services (declarative, procedural), on rules of cooperation (parallel execution, sequential execution), on middleware services (message routing,

* This work has been supported by the grants of Czech Science Foundation No. 201/02/1456 and 201/03/0911.

central services), on durability of communication partnerships (permanent cooperation, transient cooperation), and on the communication type (synchronous, asynchronous, datastore based, batch).

The practice indicates that it is good to build SOSS so that the components have character of permanently always-available activities (services). A direct consequence is that the services behave like peers of a (virtual) peer-to-peer network. It implies a limited usability of 'central' services like central databases and some properties of the interfaces of the components. We say that such systems are *service oriented* (SO) or that they have *service-oriented architecture* (SOA). Such criteria are fulfilled e.g. by systems using web as 'middleware engine' [1]. In that case the components are often integrated as web services. Two main variants of service-oriented systems are to be distinguished:

Alliances: Highly open systems having the properties that the partners for communication must be looked for. Such systems must use generally accepted and accessible middleware (e.g. Internet) and standards for message formats and protocols. Alliances are typical in *e*-commerce. The services in alliances are usually web services.

Confederations: Systems being more or less dedicated to support an enterprise or a cooperation of some well specified collections of subjects. Services in confederations are often wrapped (equipped with proper gates) legacy systems and/or third party products. Confederations have – especially in the case of proper interfaces of the services – many software engineering advantages (modifiability, stability, etc.). Confederations are more structured than alliances. It offers a better (but still limited) opportunity to model them. Preferable software engineering properties of confederations depend havily on the quality of service interfaces. The interfaces should be declarative and user knowledge domain oriented.

2 When Service Orientation is to Be Used

The globalization of organizations led to requirement of integration of legacy systems and third party products. Large on-line systems cannot be switched off for a longer time. If a legacy system is to be integrated as a component of a system, it usually must be integrated 'almost unchanged'. The only admissible change is the addition of gates (ports) providing interface functions allowing other components to communicate with the given one. Examples are:

1. *e*-government (peers are cooperating information systems of offices, e.g. information system of customs offices),
2. large enterprises (cooperating IS of autonomous divisions),
3. process control (cooperating drivers of I/O devices),
4. military systems (cooperating IS of military bodies, services, weapon technologies, soldiers),
5. systems supporting *e*-commerce (alliances).

The systems in points 1 through 4 are confederations. Confederations must be used in many other situations, general large systems inclusive. The third and fourth examples show that 'service' does not always mean 'information providing service' nor web service – a service can be arbitrary properly encapsulated activity. The middleware need not be www-like. It is even possible that some parts of the middleware are based on TCP/IP protocol, and others are implemented using operating system's tools or using shared data stores. Some of the services can be provided by 'properly wrapped' human beings. The communication between the services can be message-driven or data-driven (i.e. via a datastore).

SOSS's give great opportunities to provide functions of new types and can provide many software engineering advantages:

- large systems often must have the service-oriented architecture [2] to be developable;
- modifiability (changes tend to be local, new components can be easily integrated, different SOSS can be easily interconnected, etc.);
- large and complex systems built using SOA need less effort [2] to be developed than systems built monoliths (i.e. one application only or all the components forming the system are white boxes);
- openness (easy to connect other systems or integrate new services);
- support for selective reengineering and outsourcing (some services only);
- simplified restructuring of the organizational structure of enterprises (e.g. decentralization);
- simplified business process reengineering (BPR) and the applicability of agile forms of development for the construction of big systems;
- the chance to exploit extreme/agile programming in large projects.

3 What Interface for What Purpose: User Point of View

Until now we have discussed the software services (i.e. permanently active processes) from the software developers point of view: as black boxes that are nodes of a virtual peer-to-peer network and have specific interfaces The question is how the service interfaces should look like.

It is known that the main reasons for software project failures (compare e.g. [3]) are the problems with requirements specifications. The corner stone of the specification of service-oriented systems is the specification of service interfaces. The interfaces of service should therefore be understandable to users. Due various reasons (e.g. the universality of the used tools, the necessity to use standards) the interfaces of the services in alliances must be quite programmer-oriented (procedural). It to a high degree limits the user involvement in the design and implementation of service interfaces. This is a crucial disadvantage.

The practice has shown that the communication between services in alliances should be stateless. It has a substantial advantage for the programming of the services, as they can be to a high degree designed and written like the classical programs with terminals. The very flexible structure of alliances is very difficult

to model. This is the reason why we shall discuss mainly the case of confederations where the interfaces can be user-oriented. Such interfaces tend to be of high level (declarative) and the communication tends to be coarse-grained and declarative rather than procedural. In most cases the software services coincide with well defined real world business services (or other human activities) and their interfaces are based on coarse-grained (declarative and semantically rich) commands reflecting the structure and semantics of commands used by human beings. As real world services change rarely(compare bookkeeping), there is a good chance that the interface of the corresponding software services will not vary for a quite long period of time. This is obviously a substantial advantage.

The user-oriented communication between communication parties can be logged and analyzed quite easily as the semantics of the messages is clear and usually there is not too many messages (as the messages are semantically rich and the communication is coarse-grained). Such communication need not be influenced by frequent changes of XML-based standards.

User-oriented interfaces can be specified by users together with developers in the way known from agile programming [4]. Interfaces specification can be the only document specifying the service. It can hide implementation details of the service. This property substantially increases the stability of the system. User-oriented interfaces simplify creation of screen prototypes replacing the services not developed yet, simplify the analysis of system failures, and are advantageous in lawsuits in the case of failures of mission critical systems [2].

These advantages are paid for. User-oriented messages will have formats that are too specific to be (within a reasonable time) standardized. This problem is not too strong, provided the communicating parties are in a long-term partnership. User-oriented declarative interfaces are semantically complex, dedicated, and sometimes data-driven. It is therefore difficult to model them in details.

3.1 Procedural or Declarative Interfaces

SOAP and related XML based standards (WSDL, UDDI) are influenced by object-oriented like philosophy. It is manifested by the use of remote procedure calls. These standards are general in the sense that they define a very wide class of interfaces. They must therefore be procedural and programmer oriented. It is a barrier for the user involvement in requirement specifications and it is not preferable if a high-level declarative user-oriented communication requirements specification must be translated into SOAP messages. The developers of confederations must in this case transform declarative-type commands into sequences of SOAP-protocol steps. This is source of errors and delays. These disadvantages can be weakened by the application of a compiler-like tools that simplify the transformation of declarative messages into sequences of SOAP messages (SOAP 'programs'). SOAP increases communication traffic and can overload XML parsing tools. SOAP is too implementation oriented. It tends to disclose implementation details. It decreases the system stability and modifiability and substantially increases the problems of reengineering. There are effectiveness

problems. A solution of these problems is the concept of front-end gates (FEG; see Fig. 1) discussed below.

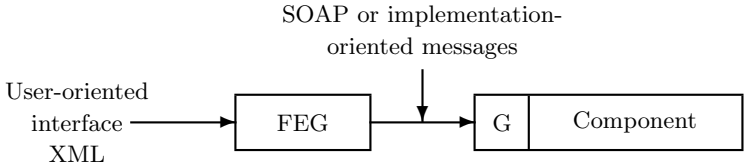


Fig. 1. Access to a component providing some service

3.2 Messages or Data Stores

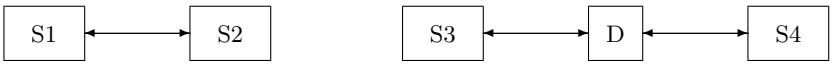
The communication via messages is preferable if services should collaborate 'on-line' (immediate response is necessary) and the services support the operative level of the activities of the users. On-line (interactive) communication can/must sometimes be implemented via datastores, using a database with triggers. We shall call such datastores *interactive datastores*.

The communication can be also implemented via a database without triggers. The datastore is then actualized periodically in batch mode. In this case the addressee (destination service) can use quite intelligent techniques to deduce the actions it should perform ([5]). The message acceptance procedure can consult human beings to find the best solution. We shall call such datastores *batch datastores*.

Batch datastores must often be used to support user management activities. Management activities often need not have 100 percent actual data to make a proper decision. The available data need not be complete even in the case that they are updated on-line as some data are not available at given moment. Many substantial internal data are hidden in the minds of people or are not collected as they are rarely used. Sometimes the data are more or less intentionally spoiled (compare [6]).

The use of batch datastores, provided that they are applicable, has substantial advantages, as it increases the autonomy of the communicating services and substantially reduces development effort (the ratio 1:5 is not an exception – Kaláb, CEO of BERIT Software, Brno, private communication, 2004). The high autonomy of the services communicating via a batch datastores increases the system stability, openness, and flexibility, and reduces development as well as maintenance effort. For example, a flexible manufacturing system thanks to batch datastore communication with the enterprise level has been used without substantial changes and/or maintenance for more than 20 years. Note, however, that the decision to use batch datastore as communication tool must be made on a careful feasibility analysis and included in requirements specification.

Both variants of communication can be modeled by fragments of diagrams from Fig. 2. Note, however, that if a service S is an absolute black box – it is, if it cannot be modified at all, its interface inclusive, the communication of it by a datastore D is more flexible as D is accessible from many services. The



A: Command interface B: Communication via datastore

Fig. 2. Command and data driven communications

main disadvantage of the data-oriented communication can be problems with the timeness of the responses of the services. The need for timeness is often overestimated.

Further advantages of data-oriented communication are: easy construction of prototypes, easy development of particular services, and the possibility to use the batch techniques for some services [2, 5].

Note, that the links in Fig. 2 are virtual. Statically the service-oriented systems can have the structure from Fig. 3.

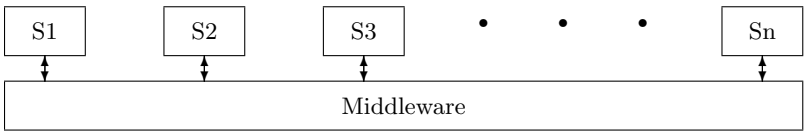


Fig. 3. Static structure of a service-oriented systems. Middleware can provide on-line as well as data oriented communication.

In Enterprise Service Bus by Sonic Software is the architecture from Fig. 3 generalized such that the middleware behaves like a bus (see Fig. 4). In the bus-

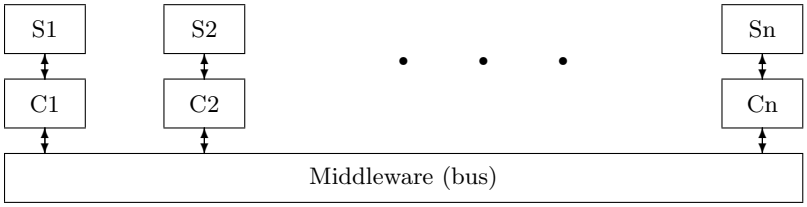


Fig. 4. Bus-like middleware with connectors

oriented architecture some specific components (connectors) that can be services are inserted between the services and the middleware. Connectors transform messages and can route them. Connectors have a lot of common with a more general concept of front-end gates discussed below.

4 Front-End Gates and Infrastructure Services

Bus-like middleware is difficult to use in loosely coupled systems like the *e*-government, military systems, the systems supporting collaboration of health care institutions, and in many other cases. In these cases the collaboration of the constituent services is quite dynamic and the services should have user-oriented interfaces in order to make the interfaces stable and understandable for

users and their experts. It is necessary especially in the case when it is required that the business people should be responsible for their processes – even during the court trials. It is also good for emergency responses.

The system developers and users should have a tool to transform message formats and for data formats using e.g. XSLT to make them declarative and adapted to the needs or requirements of the communicating partners. The solution is the use of front-end gates [7].

A front-end gate (FEG) is an infrastructure service (it is a peer of the virtual peer-to-peer network having a special functionality) performing message or data transformations and routing. In a more general case it can perform messages decomposition and composition. Messages or data to or from a service (called in the sequel *application service*) should go via a service FEG. Any application service can have more than one FEG (Fig. 5). The concept of front-end gates can

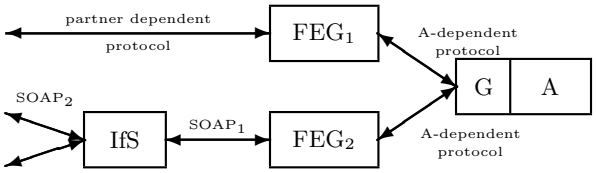


Fig. 5. Part of the structure of a confederation

be enhanced to obtain so-called *infrastructure services* (IfS). IfS can accept tuples of messages and data from different sources/services, compose the messages and data into new messages and sent the resulting messages and data to different destination services. The infrastructure service should be connected with other infrastructure services and front-end gates only.

It is also good to design the portal(s) of the system as specific service(s) (Fig. 6). Portals can use front-end gates like application services services. The

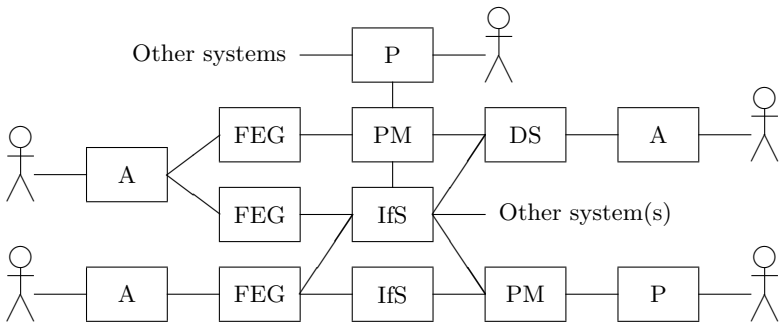


Fig. 6. System with portals P, applications A, front-end gates FEG, datastore DS, process managers PM, and infrastructure services IfS. Note, that an application service can preserve its own original local user interface. IfS, FEG, and PM are white boxes.

portals (P) and application services (A) and can communicate with users or real world processes. They can therefore have a very long response time. It is not the case of infrastructure services (IfS), data driven communications (i.e. communications via the datastore DS) and front-end gates (FEG). PM (process manager) is a specifig FEG discussed below. Here we can see certain similarities with class-object diagrams in UML, data-flow diagrams (it contain the dastore DS), and diagrams of coloured Petri nets (IfS, FEG) – compare Fig. 6.

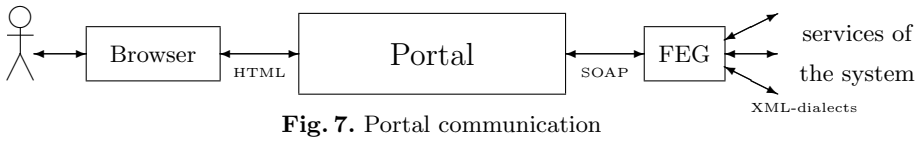


Fig. 7. Portal communication

5 Business Processes

Service orientation is based on the concept of mass parallelism. Service-oriented software systems (SOSS) must, however, support (business) processes the steps of which are actions of the software components providing software services in the above sense. Software processes are networks of actions. It is, the processes are not stateless. The open problem is how to implement the business (and workflow) processes in SOSS. We have (like in human society) the following possibilities:

1. There are no data outside the services defining the process and its states. The process is encoded into behaviour patterns of the services (i.e. each service alone 'knows' how to support the process). This is the case of alliances. The main engineering disadvantage is that the changes of business processes must be implemented via modifications of services being black boxes.
2. The states and the structure of processes are included in messages/data stores. This model is more flexible but the changes of processes must again be implemented by services (like in point 1).
3. There is service (process manager) defining the process and storing its state(s). It is desirable that the service is no central service of the whole system. The best solution seems to be to generate for every business process a new control service called process manager (PM) for every new business process. PM can be on-line modified by human management if necessary.

PM can use models and data borrowed from workflow systems. It is also possible to use some (slightly modified) diagrams of UML (activity diagrams and generalized sequence diagrams). A new service S of the type PM is generated during the instantiation of a new business process P. S can communicate with the owners of P via a portal or directly. S can use different ways of the implementation of the control data of P and its state (business rules [8], workflow models [9], UML activity diagrams, simple fulltext description). The owners of the process should be able to generate the data and or to modify them in emergency situations. It is a case of end-user development (see Comm. of ACM, issue

on End-User Development, September 2004). Such requirements are difficult to met if the manager is implemented via a central database. Central database of process models can be however used during the process instantiation (PM is here a service oriented implementation of the principles from [8]). PM then behaves like a front-end gate of the portal.

6 Diagramming of service oriented systems

The structure of confederations can be quite complex, dynamic, and the relations between the services quite rich. The functionality of application services is hidden behind a semantically rich declarative user oriented interface. Infrastucture services can perform complex message formats transformations (compare Fig. 5). All this implies limits of the power of models of SOA systems.

The modeling tools of SOSS should use static diagrams showing the static relations between software components. An example is shown in Fig. 4.

The logical (and dynamic) structure of SOSS can be modeled by the diagrams shown in Fig. 6 and Fig. 5. The complexity of the structure of diagrams implies that the graphical components of diagrams can depict the global information only and should be used mainly as links to a more detailed description of a given part of the diagram.

The static model can be to some degree viewed as a logical counterpart of class diagrams (but the similarity is very superfluous) whereas the diagram from Fig. 6 could be to some degree viewed as something like object diagrams.

The semantically rich interfaces of application services probably reduce the power of the models. The models of SOSS can be probably used as hints only how the system is structured, but they are unlikely to be able to be used as generators of some substantial parts of SOSS. This will be topic of further research.

7 Conclusions

Service orientation is an important paradigm promising to solve many software engineering issues. It is confirmed by the interest of big players (IBM, MS, SAP, ...), by the success of tools supporting service orientation (see XML and its dialects), and explosively growing interest (web services). There are commercially accessible means supporting confederative architecture (e.g. Enterprise Service Bus by Sonic Software). The paradigm is, however, new for many people. No system known to the authors fully supports the concept of front-end gates and the possibility to support data driven communication properly. There are many domains where the service orientation is the only feasible solution (global systems, *e*-government). Even for smaller systems the service orientation has significant advantages, as the decomposition into services brings substantial effort savings and increase the system quality. Services can often be relatively small autonomous applications that are easier to develop [10, 11]. Their developers can use extreme programming [12] with all its advantages.

Service-oriented systems are – according to present state of art – quite difficult to model. The diagrams discussed above model the main logical properties of SOSS only. The possibilities to achieve the situation similar to the model-driven architecture used in object-oriented systems allowing to control and support the development or even to generate the system are not for SOA the matter of the near future. The main limitations of models of confederations are due the peer-to-peer philosophy complex user-oriented interfaces of services and the dynamic properties of the virtual channels between the services in SOSS.

Although the main implementation techniques applied during the development of SOSS are known for decades it is quite difficult for a typical developers to accept the service oriented philosophy requiring e.g. to use legacy systems, cooperation with users, developing ISS etc. It indicates that the service orientation is indeed a new paradigm for many software people. The main barrier seems to be the basic attitude of many programmers that can be characterized as hacker syndrome. The symptoms are dislike to contact users, programming as an intellectual game, inability to understand methods of experimental sciences, and spoiled social abilities.

Many practical and reserch issues must be solved yet, for example modeling tools and service oriented CASE systems provided 'coordinated' models of hardware links, virtual channels (Fig. 6), and something like a layer of business processes.

References

1. Watling, N.: Web services in enterprise computing (2003) Talk at SI'2003 in Prague.
2. Král, J., Žemlička, M.: Software confederations – an architecture for global systems and global management. In Kamel, S., ed.: Managing Globally with Information Technology, Hershey, PA, USA, Idea Group Publishing (2003) 57–81
3. Standish Group: The chaos report (1994, 1996, 1998, 2000, 2003).
4. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Agile programming manifesto (2001) http://www.agilemanifesto.org/.
5. Král, J., Žemlička, M.: Requirements specification and software engineering properties of service-oriented systems. In Khosrowpour, M., ed.: Innovations Through Information Technology, Idea Group Publishing (2004) 265–268
6. Goldratt, E.M.: Critical Chain. North River Press, Great Barrington, MA (1997)
7. Král, J., Žemlička, M.: Autonomous components. In Hamza, M.H., ed.: Applied Informatics, Anaheim, ACTA Press (2002) 125–130
8. Busines Process Management Initiative: Business process management initiative home page (2000) http://www.bpmi.org/; as visited in October 2004.
9. Workflow Management Coalition: Workflow management coalition home page (1993) http://www.wfmc.org/.
10. Král, J.: Informační Systémy, (Information Systems, in Czech). Science, Veletiny, Czech Republic (1998)
11. COCOMO: COCOMO II (1995) http://sunset.usc.edu/research/COCOMOII/.
12. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley, Boston (1999)