

Event-based Software Architectures¹

Monique Snoeck - Wilfried Lemahieu - Cindy Michiels - Guido Dedene

KULeuven, Department of Applied Economic Sciences
Naamsestraat 69, 3000 Leuven, Belgium

{monique.snoeck, wilfried.lemahieu, cindy.michiels, guido.dedene}@econ.kuleuven.ac.be

Abstract. Implementation architectures of today are based on the modularisation of software into objects, components, web services, (intelligent) agents, ... with communication and coordination between components being based on peer-to-peer communication (a client-component requests a service from a server-component). Because this *binary and uni-directional* form of communication implies substantial restrictions on software maintainability, this paper proposes the development of a new *N-ary and multi-directional* communication paradigm based on the notion of "event": components will interact by jointly participating in events. This new communication paradigm uses event broadcasting as a coordination mechanism between software components. It can be implemented by means of generic binary interaction frameworks applicable across diverse platforms (distributed, web-based and centralised systems) and implementation paradigms (synchronous and asynchronous communication).. In addition, events can be enriched with intelligent features so as to be able to act autonomously and to be capable of undertaking some rescue actions when one of the composing actions fails.

1. State of the Art

Today's software architectures are based on the modularisation of software into components. Depending on the level of granularity these are called *objects*, *components* or *packages* [1]. If they act over the Internet they are denoted as *web services* [2]. Furthermore, if they are enriched with some intelligence mechanism they are called *Intelligent Agents* or *Autonomous Software Agents* [3]. In the remainder of this text we will use the term "component" as a generic term to capture all those alternatives.

In the state of the art of software development, the prevalent interaction scheme between components is based on the concept of *requests between peer components*. Each component publishes an interface listing the services other components can request from it. Collaboration between components is achieved by having one component calling the services of another component. Such a call is called a service request or (in the case of objects) a message. In many situations however, the requesting component (the sender) needs the services of more than one other component in order to achieve a particular task. In that case a message should be sent to many receivers and their answers need to be coordinated. In such a case a notification and

¹ © Springer-Verlag. This paper will be published in the proceedings of the OOIS'03 conference in the Springer LNCS-series : <http://www.springer.de/comp/lncs/index.html>

coordination scheme needs to be conceived to ensure that on all relevant components the correct service is requested and that they act accordingly. Today, the notification schemes are developed in an ad hoc manner; there is no standardised way to deal with multi-directed messages or requests.

A major problem with the use of ad hoc notification schemes is that it leads to systems that are more difficult to maintain. The maintainability problem stems from the fact that the number of possible notification schemes explodes faster than $N!$ in terms of the number N of components that need to be notified. Fig.1 shows the basic notification scheme for two components: one component is the notifier and the other component is the one that needs to be notified. In terms of message passing it means that one component is the sender of the message and the other component the receiver. In the case of N components ($N > 2$), one of the components is the notifier and the remaining $N-1$ components must be notified. The sender component can notify all the receivers itself or it can delegate the notification to one or more receivers. Fig. 2 shows the notification patterns for $N = 3, 4$ and 5 . In addition, for each pattern, there are in principle $(N-1)!$ different possibilities to assign the roles to each of the $N-1$ receiving component. For example, for $N = 4$ there are 5 different patterns, having each 6 ($= 3!$) different assignments of the roles.

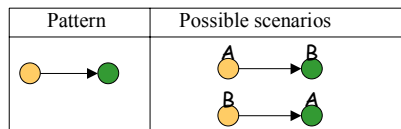


Fig. 1. Basic Notification Pattern

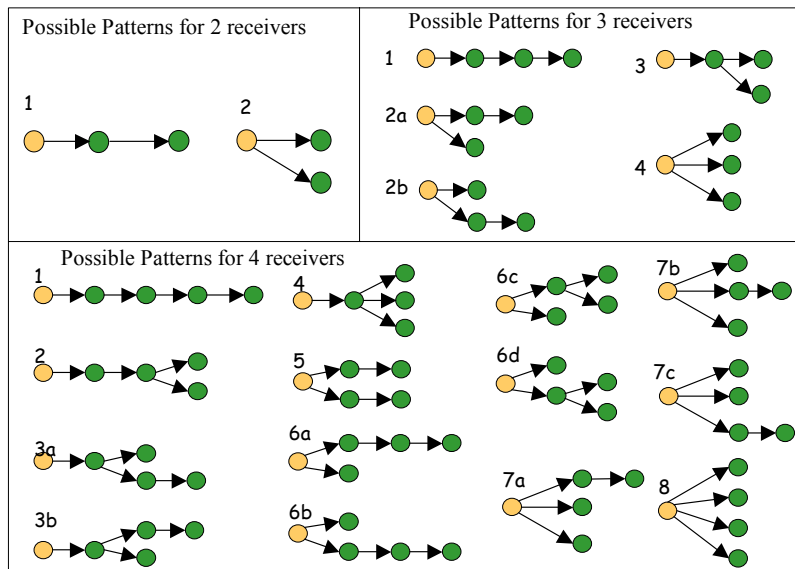


Fig. 2 Notification patterns for 2, 3 and 4 receivers

Generally speaking, the functioning of object oriented software can be described as a set of notification schemes linked to each other. For example, User Interface objects notifying transaction objects, which in their turn notify database objects. Also at a larger scale or in distributed systems, the functioning of the software can be described as a set of components linked to each other by means of notification schemes. The multitude of notification patterns makes understanding, maintaining and adapting software a difficult task.

2. Towards enhanced flexibility and adaptability of software

The boat-story of M. Jackson in [4, p. 9] aptly illustrates that the adaptability of software is proportionate to the degree to which the structure of the solution domain, in casu the structure of the information system, mirrors the structure of the problem domain. Today, it is well agreed that domain modelling is an important element in the development of adaptable and flexible information systems [5]. Most enterprise information systems indeed contain objects and components that mirror elements of the real world. For example, an insurance system will have components that represent customers and policies because in the real world their business is about customers and policies. Whereas a one-to-one mapping can easily be established between the data-aspects (that is to say, the static aspects) of an enterprise information system and concepts of the real world, this one-to-one mapping is much more difficult to establish between the behavioural component of an information system and the real world. The modelling of the dynamic aspects in an information system is restricted to the definition of services in the context of a single component, whereas in the real world events encompass actions in many components. For example, by ordering a product, the stock-level is adjusted and an order is created. In the real world, events are just there, they simply happen and are the basis of the dynamics of the real world [6]. In order to mimic the real world, the enterprise information system should also contain a model of these events.

This paper proposes a component interaction mechanism based on the notion of event. The underlying hypothesis is that at the conceptual level, the concept of "event" can be used to denote something that mimics real-world events and that encompasses actions in multiple components. In the information system (the solution domain) the effects of an event on real-world entities are implemented as a set of corresponding actions which can be procedures in individual objects just as well as services of components or web services. A major difference is however that, whereas in the real world events can occur without human intervention, in an enterprise information system, some human intervention is required to notify the information system that an event has occurred. Usually this is done by means of some user interface component. As a result, an event in an information system is always triggered by some component and can be seen as the conceptual equivalent of a multi-directed message from the triggering component to the executing components.

At the implementation level, we assume that it is possible to mimic a real-world event by means of a piece of software, the so-called *event dispatcher*, that is responsible for dispatching the event notification to all participating components and that is able to coordinate the responses (success or failure) and provide the triggering com-

ponent with feedback about the execution status. By implementing a real-world event directly as a unit of component interaction the structure of the solution domain better reflects the structure of the problem domain as represented in Fig. 3. Notice that the detection of the occurrence of a real-world event is the responsibility of the triggering component, unless the event is enriched with some additional intelligence making it able to execute spontaneously.

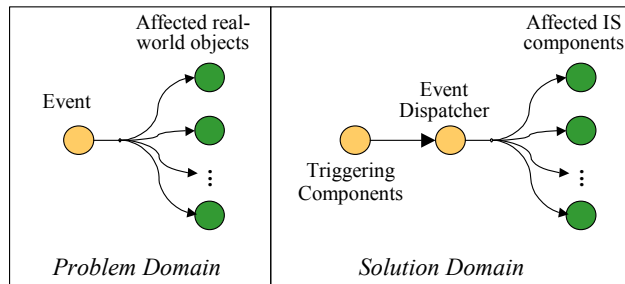


Fig. 3 Improved structural correspondence between problem domain and solution domain

The development of object interaction by means of multi-directed messages is a research project at the Management Information Systems group at the KULeuven. It has not been realised in its entirety yet but is still research in progress. In section 3 we present the major architectural principles and demonstrate that the communication mechanism is universally applicable. We then present in section 4 current implementations of the concept and the experienced advantages in terms of maintainability.

3. Architectural Framework

In order to ensure the universal applicability of the event concept, the event dispatcher will be developed as an independent software element, capable of dispatching to and coordinating the responses of the receivers. In this way an event becomes a "service" that is jointly delivered by multiple components and that can be used by any component needing this service. The name of the event dispatcher will be the name of the *event* upon which the dispatching service is triggered. In that way we can say that the *event dispatcher* is the implementation of the conceptual *event*.

An event dispatcher can also be seen as the implementation of a service of a component, where the component needs to dispatch the service request to its constituent components. The services of the constituent components can at their turn be defined as coordinated actions implemented by means of an event dispatcher. In this way event dispatching can be recursively defined across levels of granularity (see Fig. 4).

From an architectural point of view, the main research question is how exactly to design this piece of software, such that it is a general pattern applicable

- over all different software platforms (languages, environments),
- both on a single platform and in a distributed environment (which web services are an example of),
- at all levels of granularity.

The event dispatcher must account for the fact that one of the executors might not be able to perform the requested action. In that case the event dispatcher is responsible for ensuring consistency. If it is required that either all executors or none of them execute their corresponding action, in case of failure for one of them, the other participants will have to roll back their already performed action.

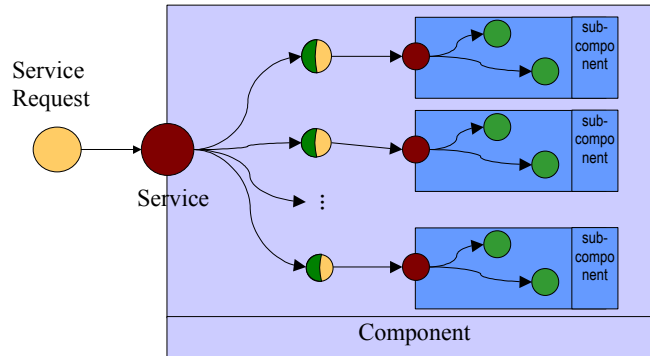


Fig. 4 Recursive usage of event dispatching

Unlike existing transaction mechanisms that expect a transaction to take place in a very limited amount of time (such as commit and rollback features in a database environment), an event should also accommodate for *long lasting coordinated actions*, as frequently seen on the Internet. For example, the event "book a trip" is composed of the actions "order a plane ticket", "book a room in a hotel" and "reserve a car". Confirmation of the composing actions can take 24 hours or more.

In addition to the enhanced possibilities for coordination, events also offer the possibilities to add intelligence to object interaction. A recent evolution is the development of Intelligent and Autonomous Agents as a means to overcome the passive nature of software [3]. An intelligent component should know about its goals [2], should be able to take initiative and to undertake actions in order to fulfil the goal and be able to try out different scenarios rather than failing immediately when one of the actions fails. Such a goal-oriented approach would allow to better address the strategic dimension of information systems [7]. Applying the agent concept to events offers interesting perspectives to improve the choreography of multiple components that together perform a complex task. A long-lasting coordinated action is not necessarily based on a single scenario that is either executed completely with success or fails at some point. An "intelligent" event validates a request for execution beforehand, for example by means of contracts [8, 9] and should be capable of undertaking some *rescue actions* when one of the composing actions fails. For example, an intelligent version of the "book a trip" event described above, would try an alternative car-rental company in case the initial "reserve a car" does not succeed.

Secondly, whereas current software is mostly composed of passive components that wait for some external component to trigger the execution of a method, adding intelligent *autonomy* to an event would mean that a triggering component is no longer required for an action to take place: events should be able to execute spontaneously on the basis of a set of business rules. As an example, let us consider a typical busi-

ness example of handling sales orders. When a customer orders something, a sales order is created with one order line per ordered product. Suppose that an order is not always delivered at once but can be the subject of multiple deliveries. In addition, a business rule states that a sales order can be billed only if all the products have been delivered. In a classical approach the set of pending sales orders must be checked periodically to see which ones can be moved to the billing state. This amounts to a service in the enterprise information system that is executed either manually or initiated by a timer and that triggers a billing event for each sales order that is ready to be billed. As a result, there will always be some delay before an order ready for billing is effectively billed. In an "autonomous event" approach, the billing event should be equipped with business rules defining when the event should occur. As soon as all the rules are satisfied, the event is executed automatically. For example, as soon as the delivery of the last product of an order is registered, the billing event for that order can execute itself spontaneously.

4. Current Realisations

a. Standalone environment

The principle of events as multi-directed messages has first been implemented in the context of MERODE [10, 11] for a standalone environment. MERODE is a domain model approach in which an enterprise or domain model consists of enterprise objects and business events. Enterprise objects synchronise with each other by jointly participating in events. In the implementation architecture, a business event is implemented as a class with a method for checking whether all business rules associated with the business event are satisfied and a method for dispatching the event to all involved enterprise object classes. Information system functionality is modelled as a layer on top of the enterprise model. An information system is hence constituted of two layers: a business layer containing the enterprise model and an information system layer containing the information system services on top of it (see Fig. 5). In the Information System Layer, information system events will trigger input and output services. Each information system service that needs to modify information in the enterprise layer can only do so by using services provided by the events. One of the major advantages is that a single event can be triggered by different information system services and that all intelligence related to validating the event against business rules is located in one place. For example, in a banking system, the enterprise model would contain enterprise objects such as customer and account and a business event such as withdraw. The withdraw-event knows about all the rules pertaining to withdrawal of money. This event can now be used by all services allowing to withdraw money such as the counter application and the ATM application. Output services relate to information that is extracted from the business objects and are implemented by means of a set of corresponding attribute inspections (e.g. by means of a query). As attribute inspections do not involve changes in a business object, they are not the subjects of transaction management. In contrast, input services relate to modifications of business objects and are therefore only allowed by means of the intermediary of a

business event. The handling of concurrent business events is subject to transaction management.

In adhering to this basic layering and to the principles of object-orientation, a flexible architecture is obtained, because input and output services can be plugged in and out of the Information System Layer with no side effects on the objects in the Enterprise Layer.

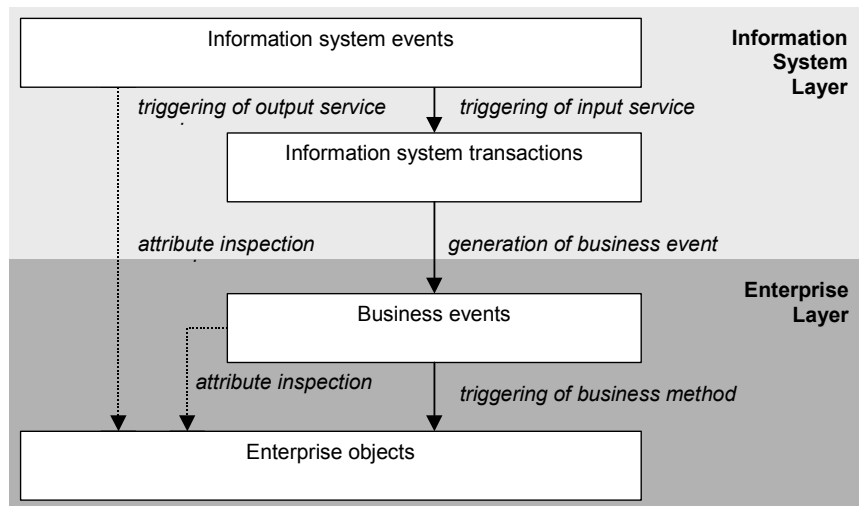


Fig. 5. Basic layers to sustain event-driven development in a standalone environment

b. Distributed and strongly-coupled environment

The architecture presented in the previous section can easily be extended to *intra-enterprise* application integration based on a LAN based distributed object framework, e.g. [12].

In such a case, the Enterprise Layer is used as an integration layer for off-the-shelf business support or operation support systems (BSS/OSS). The Information System Layer now consists of independent BSS/OSS and possibly self developed information services (see Fig. 6). The event dispatchers now take the role of coordination agents. Each BSS/OSS uses the services of the event dispatcher whenever it wishes to update information in the Enterprise Layer. When a BSS/OSS wants to be notified of updates in the Enterprise Layer, it can subscribe to an event dispatcher in order to be notified upon occurrence of the event. In this way, the BSS/OSS can keep its own information synchronised with the information contained in the Enterprise Layer.

Compared to a classical stove-pipe approach where BSS/OSS directly interact with each other, the integration approach based on an Enterprise Layer offers better maintainability and flexibility [12, 13]: each BSS/OSS can easily be removed and replaced by other software without affecting the other BSS/OSSs. In addition, the event-based coordination mechanism allows easily to deal with different levels of API-functionality offered by the BSS/OSS. For example, if the API-functionality offered by the BSS/OSS is insufficient to let the BSS/OSS call the event dispatcher, the event

dispatcher can be equipped with a "listening" functionality, allowing it to detect event occurrence in the BSS/OSS.

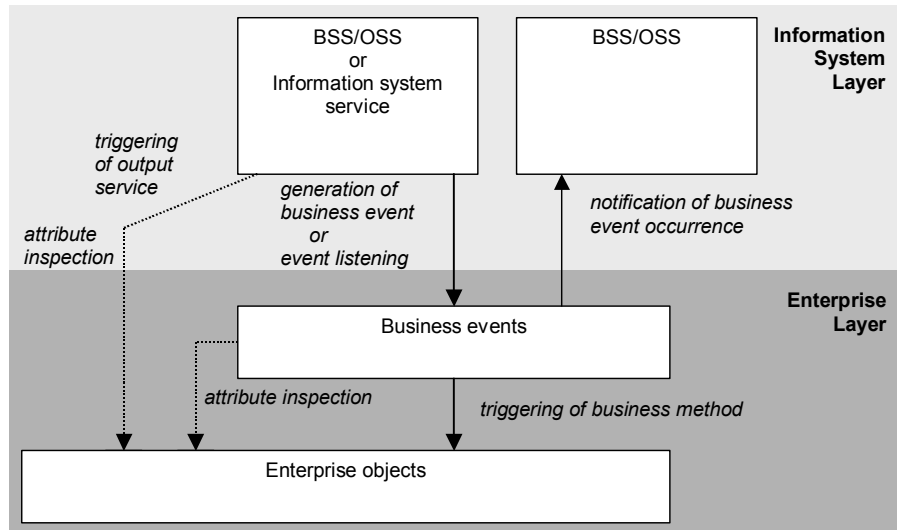


Fig. 6 Event -based architecture for enterprise application integration

c. Distributed and loosely coupled environment

Both in the stand-alone environment and in the distributed and strongly-coupled environment, there is a common *Enterprise Layer* that contains the “primary copy” of all enterprise objects. In such a case, the Enterprise Layer can be considered as a (set of) component(s) offering two types of functionality: *attribute inspection* and *event notification*. An analogous way of working can be used to define a distributed and *loosely-coupled* architectural model targeted at a web services environment, where the Enterprise Layer itself is distributed among multiple sites, possibly developed and controlled by different authorities [14, 15]. The enterprise model can now be considered as the enterprise model for an *extended enterprise*, which may eventually be implemented by multiple, independent parties to support *inter-enterprise* application integration².

In a LAN based implementation, “real world” events are translated into business events in the information system by means of so-called *input services*. In general, these will encompass user interface components that receive input from human beings, e.g. the “sign” button in a sales order form. The business event is then broadcast by the *event dispatcher*. Now each web service may have its own local input services. Events can be dispatched to the service’s local objects. However, the assumption of a predefined set of *remote* objects to which a certain event may be of interest is unrealistic: in many cases, web services are developed without prior knowledge of the other services they are to interact with, and certainly without knowledge about these ser-

² However, the architecture is no less suitable to enterprises that interact in an ad-hoc manner, i.e. without a “unified” business model for the extended enterprise, as discussed further on.

vices' internal enterprise objects. Therefore, as to remote services to which the event may be relevant, the approach should cater for an explicit *subscription* mechanism. A given web service's event dispatcher will dispatch its events to all local objects *and* to all remote services that are explicitly subscribed to the corresponding event type. Subscription by a remote service comes down to a local *stub object* being created, which locally "represents" this remote service and contains a reference to its URL. The resulting event based interaction mechanism takes place in four stages, as illustrated in Figure 7. First, the input service associated with a web service detects a real world event (1). This event is broadcast by the service's event dispatcher to all *local* objects, among which some stub objects (2). Each stub *propagates* the event to the remote service it represents (3). Each remote service's event dispatcher in its turn broadcasts the event to its own local objects (4). The appropriate (local and remote) objects then each execute a corresponding method, in which preconditions are checked and/or updates are executed.

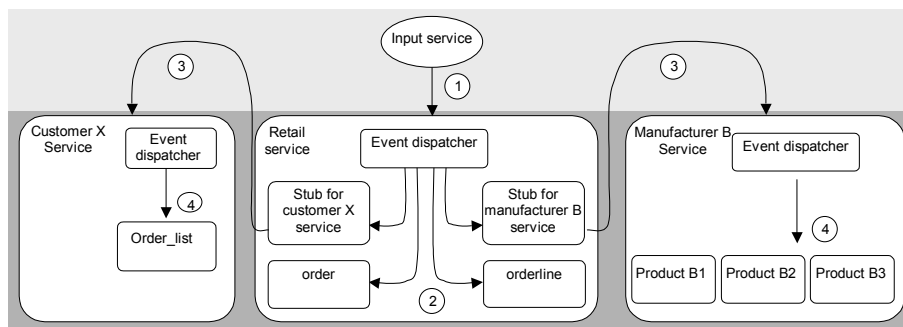


Fig. 7. Example of an event-based architecture for web services

Hence the event based interaction mechanism can be applied on two different levels in the web service architecture: in the first place for interaction *between* web services, where each web service is perceived as an atomic unity by its peer services. The services will interact by responding to communal events. A second interaction mechanism exists at the level of the *intra-service* interaction, i.e. between the respective enterprise objects that make out a single service. This approach can easily be generalized into an N-level system: a service or component receives an event notification and propagates it to its constituting components (and, through the stubs, to the appropriate remote services), which in their turn propagate it to their components etc. In this way, the event is propagated recursively at each level in a hierarchy of complex web services with a complex task, that in their turn are conceived of more simple services with a simpler task until a level of atomic services is reached. On the other hand, at each level, a component that receives an event notification may be a "wrapper" that internally consists of components that interact by means of another mechanism.

The event based interaction mechanism can be applied to web services that belong to long standing business partners with more or less established interaction patterns and to partners that participate in short lived, ad hoc partnerships. In the first case, one could start out from a "unified" analysis and design over the extended enterprise, resulting in a "unified" business model that entails the enterprise objects of all partners involved. In this approach each web service, from the moment it is deployed,

has a stub object for each remote service it interacts with. As to ad hoc interaction, stub objects will be created at runtime, as the consequence of certain events occurring. Such events will represent the explicit *subscription* of one service to another. From then on, two services can start interacting, through mediation of the newly created stub. In a similar way, events that induce the deletion of a stub object terminate the interaction between two services.

5. Discussion

The main research goal of this project is to alleviate the maintenance problem caused by one-to-one interaction by developing a universal communication paradigm that is able to handle notification and coordination when multiple software components need to interact upon occurrence of an event.

In terms of maintainability, the event dispatching pattern is much easier to adapt than arbitrary message passing patterns. The main advantage is that the dispatching pattern is fully independent from the number of components to which the event must be dispatched. As a result, adding or dropping a component does not cause any redesign of component interaction as opposed to the classical approach [16]. In addition, the dispatching pattern can easily account for distributed components, either tightly or loosely coupled.

Event dispatching can be implemented by means of all known message passing mechanisms. For web-services it can be implemented through simultaneous SOAP messages, i.e. in full compatibility with the current standard web services stack. Whereas the web service concept in itself already entails a loose coupling mechanism, the coupling between web services communicating by means of event broadcasting can be kept even looser, e.g. the number of parties that participate in an event can be easily increased by just adding another service that subscribes to the event, without having to redesign the entire chain of one-to-one message exchanges.

As already discussed previously, event propagation can be used both for interaction between peer services and for a complex service to co-ordinate the behaviour of its components. The clear distinction between attribute inspections and business events allows for focusing on only the latter with respect to *transaction management*. Also, transaction specification is simplified: a single business event results in multiple simultaneous updates in multiple enterprise objects. The latter is much easier to describe than the myriad of one-to-one message exchanges that could make out a single business transaction in a pure message passing or RPC-based approach.

Since current object-oriented languages offer no native language support for events (at least not as a mechanism to implement co-ordinated actions) future research aims at the development of mechanisms for the realisation of this communication paradigm. This should result in design patterns and/or frameworks that can be used for all types of environments. A combination of these frameworks and patterns with tailoring facilities will be at the heart of a code-generating facility allowing software engineers to generate full implementations of coordinated actions. Native language support for the concept of event is the ultimate goal, for which the frameworks should form a solid basis.

References

1. Cheesman J., Daniels J., *UML Components, A simple process for specifying component-based software*, Addison Wesley, 2000, 208pp.
2. Fensel D., Bussler C., The Web Service Modeling Framework WSMF, *Electronic Commerce Research and Applications*, 1(2), 2002
3. D'Iverno M., Luck M., *Understanding Agent Systems*, Springer, 2001, 191 pp.
4. Jackson M.A., Cameron J.R., *Systems Development*, Prentice Hall Englewood Cliffs (N.J.), 1983
5. Nilsson, A.G., Tolis, C., Nellborn, C. (eds.): *Perspectives on Domain modeling: understanding and Changing Organisations*. Springer Verlag, Berlin (1999)
6. Cook S., Daniels J., *Designing object systems: object-oriented modeling with Syntropy*, Prentice Hall, 1994
7. Sowa J. F., Zachman J. A., Extending and formalising the framework for information system architecture, *IBM Systems Journal*, 31(3) (1992) pp. 590-616
8. McKim J., Mitchell R., *Design by Contract: By example*, Addison-Wesley (2001) 256 pp.
9. Meyer B., *Object Oriented Software Construction*, Prentice Hall, 2nd Edition (1997)
10. Snoeck, M., Dedene, G., Verhelst M, Depuydt A. M.: *Object-oriented Enterprise Modeling with MERODE*, Leuven University Press, Leuven (1999)
11. Snoeck, M., Dedene, G.: Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types, *IEEE Transactions on Software Engineering*, Vol 24 No. 24, pp.233-251 (1998)
12. Lemahieu Wilfried, Snoeck Monique, Michiels Cindy, "Integration of Third-Party Applications and Web-Clients by Means of an Enterprise Layer," accepted for Volume V of the *Annals of Cases on Information Technology*, 2002
13. Lemahieu Wilfried, Snoeck Monique, Michiels Cindy, An Enterprise Layer based Approach to Application Service integration, Accepted for *Business Process Management Journal - Special Issue on ASP*.
14. Lemahieu W., Snoeck M., Michiels C., Goethals F.: An Event Based Approach to Web Service Design and Interaction, K.U.Leuven – accepted for APWeb'03 (2003)
15. Lemahieu W. Snoeck M., Michiels C., Goethals, F., Dedene G., Vandenbulcke J., A model-driven, layered architecture for web service development submitted to *IEEE Computer*, Special issue on *Web Services Computing* (2003)
16. Snoeck M., Poels G, Improving the Reuse Possibilities of the Behavioral Aspects of Object-Oriented Domain Models, In: *Proc. 19th Int'l Conf. Conceptual Modeling (ER2000)*. Salt Lake City (2000)