Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering 51 (2004) 81–107
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Events as atomic contracts for component integration

M. Snoeck *, W. Lemahieu, F. Goethals, G. Dedene, J. Vandenbulcke

Department of Applied Economic Sciences, Katholieke Universiteit Leuven, Naamsestraat 69, 3000 Leuven, Belgium

Accepted 3 March 2004
Available online 10 April 2004

Abstract

Today many companies rely on third party applications and application services for (part of) their information systems. When applications from different parties are used together, an integration problem arises. Similarly, cross-organisational application integration requires the coordination of distributed processing across several autonomous applications. In this paper, we describe an integration approach based on an *event-based coordination paradigm*. Interaction is based on atomic units of interaction called "business events". Each business event mirrors some event in the real world that requires the coordination of actions within a number of components. The coordination between applications is achieved by having applications specify preconditions for business events. As a result, a business event becomes a small scale contract between involved applications: each application can insert its own clauses into the contract by specifying preconditions. Moreover, a formal method for *contract analysis* is proposed, to verify whether the contract is free from contradictions and inconsistencies. Finally, in addition to its contracting aspect, the event-based communication paradigm entails a *dispatching and coordination mechanism*, which offers the additional advantage of a complete separation of the coordination aspects from the functionality aspects. The paper discusses different alternative architectures for event-based coordination, with particular emphasis on distributed, loosely coupled environments such as web services.

Keywords: Event-based coordination paradigm; Business events; Preconditions; Contract analysis

* Corresponding author.
E-mail addresses: monique.snoeck@econ.kuleuven.ac.be (M. Snoeck), wilfried.lemahieu@econ.kuleuven.ac.be (W. Lemahieu), frank.goethals@econ.kuleuven.ac.be (F. Goethals), guido.dedene@econ.kuleuven.ac.be (G. Dedene), jacques.vandenbulcke@econ.kuleuven.ac.be (J. Vandenbulcke).

0169-023X/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2004.03.007

1. Introduction

Today many companies rely on third party applications and application services for (part of) their information systems. Additionally, the Internet and the web service technology enable cross-organisational collaboration. Both intra-organisational and cross-organisational application integration pose the challenging problem of creating an infrastructure supporting flexible, distributed, heterogeneous applications In this paper, we describe an integration approach based on an *event-based coordination* mechanism. This approach is deliberately kept away from a traditional document based, flow-oriented approach where business processes are hard coded into the application interaction architecture. Rather, interaction is based on atomic units of interaction called "business events". Each business event mirrors some event in the real world that requires the coordination of actions within a number of components. For a standalone application, these components are the application's own enterprise object types. For intra-enterprise application integration and cross-organisational application integration, a business event models the concurrent processing of its associated actions in the involved autonomous applications.

The use of the event-based paradigm for application integration is not really new. However, the major part of research on event-based architectures is devoted to the design of notification servers and the development of event-service middleware. Little attention is paid to event-based conceptual modelling nor to the definition of the semantics of event-based systems. More importantly, the notification in existing event-based systems is based on a pattern of "send and forget": events are anonymous, producers are not necessarily aware of the fact that their state change is observed and notified to event consumers. The possibility of return values is therefore inexistent and not even considered in the majority of event-based architectures. Moreover, whenever more than one consumer is subscribed to a certain event, existing event-based architectures assume that these consumers are completely independent and unaware of one another. As a result, the notion of "coordination" is completely absent in these architectures. It is precisely the aim of the research proposed in this paper to fill these gaps: we focus on the conceptual design of component integration and coordination and demonstrate how the use of a process algebraic description of the behavioural semantics allows to analyse coordination problems before-hand.

The remainder of the paper is organised as follows. Section 2 briefly introduces the concepts of Object–Event table and sequence constraints that were borrowed from Merode as conceptual modelling techniques for event-based coordination. Section 3 motivates the use of techniques that were developed for strongly integrated LAN-based object-oriented applications for the modelling of distributed applications composed of larger-grained components of services. Section 4 is the main part of this paper. It demonstrates how the process algebraic formalisation of component behaviour can be used to detect coordination conflicts. Section 5 outlines a number of possible notification architectures that build upon the contract verification issues presented in Section 4. Finally, Section 6 presents related work in the areas of event-based modelling, event-based architecture and process algebra's for multiparty interaction.

2. Business event-based design

The event-based interaction mechanism as proposed in this paper uses two techniques borrowed from the object-oriented analysis and design methodology Merode [41,42], namely the

object–event table and the sequence charts. Additionally, in Section 4, the Merode process algebra [13] is used to analyse coordination problems. Merode is complementary to UML [32], in that it offers a precise and computationally complete *methodology*. UML can then be used as a *formalism* to capture the Merode specifications. Merode represents an information system through the definition of business events, their effect on enterprise objects and the related business rules. Although it follows an object-oriented approach, it does not purely rely on method invocation to model interaction between enterprise object classes as in classical approaches to object-oriented analysis. Instead, business events are identified as independent concepts, with an *object–event table* defining which types of objects are affected by which types of events. For example, let us assume that the domain model for an order handling system contains the four object types CUSTOMER, ORDER, ORDER LINE and PRODUCT as depicted in the class diagram of Fig. 1.

Rather than modelling the dynamic aspects of the domain solely by equipping classes with operations, Merode recognises the fact that many business activities involve *simultaneous*, *coordinated actions* on several enterprise objects. For this reason, the dynamic aspects of the domain are modelled by identifying atomic units of action, called *business events*, and by specifying which objects are affected by which action by means of an object–event table. For the above example, possible business event types are *cr_customer*, *mod_customer*, *end_customer*, *cr_order*, *mod_order*, *end_order*, *cancel_order*, *ship*, *invoice*, *pay*, *cr_orderline*, *mod_orderline*, *end_orderline*, *cr_product*, *mod_product*, *end_product*. The object–event table (see Table 1) shows which object types are affected by which types of events and also indicates the type of involvement: C for creation, M for modification and E for terminating an object's life. For example, *cr_orderline* creates a new occurrence of the class ORDERLINE, modifies an occurrence of the class PRODUCT because it requires adjustment of the stock-level of the ordered product, modifies the state of the ORDER to which it belongs and modifies the state of the CUSTOMER of the order. Full details of how to construct such an object–event table and validate it against the data model and the behavioural model (see below) are beyond the scope of this paper but can be found in [41,42].

Each object type has a method for each event type in which it may participate. Such method implements the object's state changes (i.e. changes to attribute values) as the consequence of an event of the corresponding type.

In addition to the definition of a method-body, the enterprise objects are also able to put constraints on the events in which they participate. These are based on *preconditions* put on the corresponding event type by the object type. Special kinds of preconditions are *event sequence constraints* that can be derived from a finite state machine associated with each object type. For example when a customer orders a product, a new ORDERLINE is created by triggering the *cr_orderline* business event. Examples of preconditions for *cr_orderline* specified by the participating business objects are:

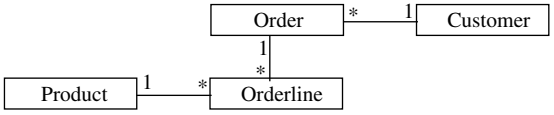

Fig. 1. Domain model for an order handling system.

Table 1
Object–event table for the order handling system

	Customer	Order	Orderline	Product
cr_customer	C			
mod_customer	M			
end_customer	E			
cr_order	M	C		
mod_order	M	M		
end_order	M	E		
cancel_order	M	E		
ship	M	M		
invoice	M	M		
pay	M	M		
cr_orderline	M	M	C	M
mod_orderline	M	M	M	M
end_orderline	M	M	E	M
cr_product				C
mod_product				M
end_product				E

- ORDERLINE. The order line should not already exist.
- CUSTOMER. A customer is only allowed to order an additional product if the resulting total price of his/her unpaid orders is not above a certain limit.
- PRODUCT. An order line can only be created if the stock for this product is sufficient.
- ORDER. An order line can only be created for a given product if the order does not already contain an order line for this product.

Sequence constraints can be modelled as part of the life-cycle of enterprise objects. Fig. 2 shows the finite state machine of the ORDER enterprise object. As long as it has not been shipped, an order stays in state 1 (modifiable). At this stage it is still possible to modify the order by adding, changing or deleting order lines. The ship event moves the order into state 2 (shipped). From then on the order cannot be modified anymore. This means that the events *mod_order*, *cr_orderline*, *mod_orderline* and *end_orderline* are no longer accepted for this order. The *invoice* event signals that the order has been invoiced to the customer. Finally, the *pay* event signals the payment of the

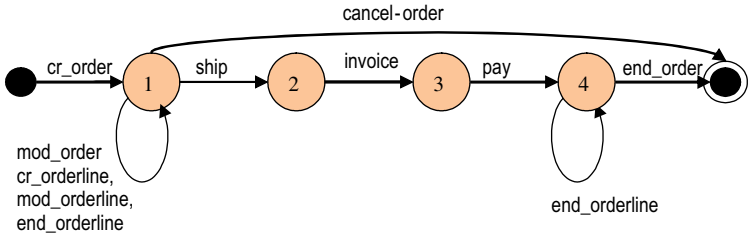

Fig. 2. State machine for an order object.

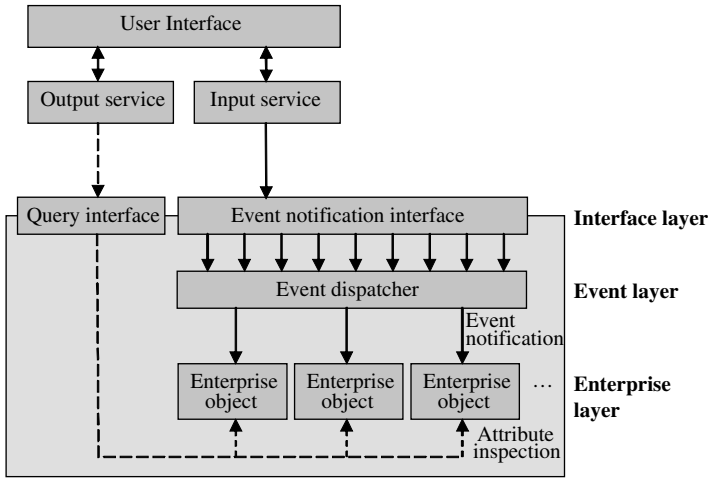

Fig. 3. General architecture of a monolithic event-based application according to Merode.

order. From there on, the order is ready for deletion. Cancelling an order is only possible in the first state, meaning that the order should already exist but that it must not have been shipped yet. The business events *cr_orderline*, *mod_orderline*, and *end_orderline* are also present in the finite state machine of ORDERLINE, stating that an orderline object is first *created*, then becomes *modifiable*, and finally is *ended*. In this way, the combined sequence constraints mimic the general business processes.

Whenever an event actually occurs, it is notified to all involved enterprise objects. Only events that satisfy the sequence constraints and other preconditions imposed by each individual object are accepted and effectively processed. For example, assume a *mod_orderline* event that raises the quantity of ordered goods. This event will need the approval of the involved ORDERLINE (existence check), the involved ORDER (sequence constraint), the involved CUSTOMER (check of resulting total price against limit), and the involved PRODUCT (check of available stock). Only if all four objects agree with the event, the corresponding object–event methods will be actually invoked and executed.

The entirety of all enterprise objects and business events that together shape the business process(es) is called the *enterprise layer*. The eventual information system is realised as a layer on top of the enterprise layer, consisting of output and input services. *Output services* use attribute inspections to query the enterprise objects and deliver the information to the user. Upon occurrence of a business event in the real world, *input services* collect input data from the user and invoke the corresponding event to update the set of enterprise objects. The general architecture of a system developed according to these Merode concepts is shown in Fig. 3.

3. Business event-based coordination

Although the Merode approach was initially developed for LAN-based monolithic applications, its concepts also suit the problem of application or component integration very well. The

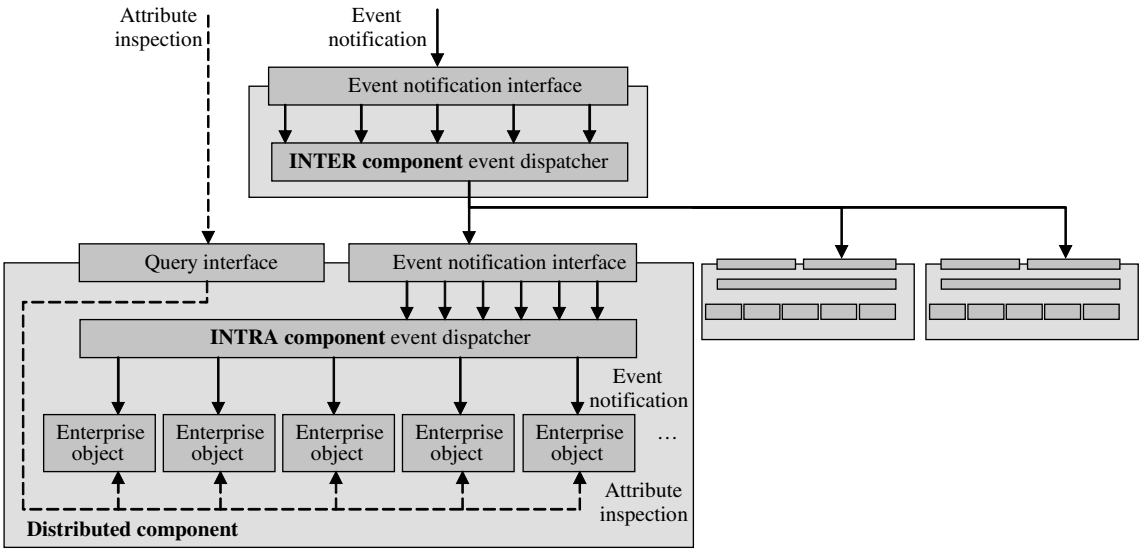

Fig. 4. Inter-component and intra-component event dispatching.

latter calls for a distributed, loosely coupled environment, where the enterprise layer itself is distributed among multiple components, applications or web services, possibly developed and controlled by different authorities. Such components [1] are coarser grained than the actual enterprise objects, as it would not be feasible for a company to publish individual interfaces to its individual enterprise objects.

Hence in a component integration context, event-based interaction occurs at the level of components; each such component is a black box to the outside world and may encapsulate multiple internal enterprise objects. If a component's internals are also conceived according to the event paradigm (which is not a requirement), it has a similar layered structure: its "local" enterprise layer encompasses several enterprise objects, which incorporate the actual business logic. The component offers a unified public interface to its (hidden) enterprise objects, which supports two types of interaction: *attribute inspection* and *event notification* (see Fig. 4). Business events are units of action that span one, two or more components and that provoke some processing in each of them. An *application–event table or component–event table* allows defining which application/component is involved in and needs to be notified of which type of business event. The implementation architecture for the event table is similar to the widely used event-based architectures for event generation and notification [8,12,27]. The component or service that triggers the event is an event producer; the components with marked participations are event consumers. The broadcasting or notification of a business event to the subscribed event consumers is the task of a so-called *event dispatcher*. The latter is responsible for broadcasting the event notification to all

[1] We will use the generic term "component". Depending on the actual environment, this component may also be a genuine application or a web service.

participating components, coordinates the responses (success or failure) and provides the triggering component with feedback about the execution status. Upon notification by the event dispatcher, the business event is processed internally by each component. If a component's internals are also organised according to an event-based interaction mechanism, the event can be further dispatched to the component's constituent enterprise objects by means of a local object–event table. This mechanism is presented in Fig. 4.

This approach is applicable both in situations where enterprises integrate their existing, independently developed components and in situations where the respective partners in an extended enterprise develop their components from a "unified" business model. In the latter case, the unified business model identifies all enterprise objects, which are then aggregated into the respective components. Determining which enterprise objects are to be encapsulated into which component depends on multiple facets, such as the cardinalities of the relationships between the enterprise objects, the types of mutual interactions (attribute inspections and event broadcasts) but also e.g. on security considerations, especially in a web services context, where the components are controlled by different authorities. The Merode methodology entails the concept of *existence dependency* [41], which superimposes hierarchical structures [2] on object interrelationships. This greatly facilitates the identification of groups of enterprise objects as candidate components or web services. The latter is an important ongoing research issue, but is considered beyond the scope of this paper.

The coordination between components is achieved by having components specify preconditions for business events. As a result, a business event becomes a small scale contract between involved components: each component can insert its own clauses into the contract by specifying preconditions. Additionally, postconditions can be used to given a guarantee on the results. Obviously, this interaction pattern easily accounts for changes in the number of components that need to be notified of an event: it suffices to equip the event dispatcher with a subscription mechanism. Whenever a component needs to be notified of a particular type of business events, it subscribes with the appropriate event dispatcher to the corresponding event. When a component subscribes to an event, it means that the corresponding cell is marked in the component–event table. Three kinds of information are associated with such entry:

- The preconditions specify the conditions that needs to be satisfied (from the component's point of view) [3] in order to accept the execution of the event.
- The actions specify the services or methods that must be called in order to notify the component of the occurrence of an event.
- The postconditions specify the guaranteed results.

When a component no longer needs to be notified of a particular type of event, it suffices to remove the subscription by removing the marked cell from the component–event table.

[2] Not to be confused with inheritance hierarchies.

[3] For applications that also adopt the event-based approach for the interaction between their internal components, the application level preconditions to an event can be derived as the combination of the preconditions imposed by the application's internal components that participate in the event.

4. Analysing contracts

4.1. Contract management capabilities

The event paradigm presented in the previous sections is in the first place a composition mechanism, which allows for the coordination of the respective components that together form a more complex system. However, as discussed in this section the role of events as *contracts* between the participating components can be an important element in service *discovery* as it allows searching for components or services that are "compatible" process-wise.

Indeed, in contrast to existing event-based architectures for event observation and notification, the business event-based coordination model proposed in this paper allows components to specify constraints on the invocation of events. By considering each row in the component–event table, we have an overview of the components involved in one event, and an overall view of all constraints imposed by the participants on this event. By comparing two or more columns we have an overview of the set of common event types for two (or more) components. As each component can define its own sequence constraints on the common (and own) event types, one has to check whether components agree on the allowed sequences for the common event types. The analysis of these constraints is an essential step in for example web service matching: it is of no use to find a web service that performs the required task if the imposed constraints are in contradiction with constraints imposed by other components involved in the same event.

The analysis of constraints can be defined as a task of the event dispatcher since the latter is the only one to have a view on all components involved in an event. In order to define the contract management capabilities of an event dispatcher, we first review the process algebra defining the semantics of sequence constraints and parallel composition of components.

As explained previously, part of the constraints imposed on events are the result of sequence constraints. Components specify sequence constraints by considering all the events that appear in their column in the object–event table. This set of events is called the "alphabet" of a component. Each component specifies allowable sequences of events from its point of view. This can either be done by means of a regular expression or a finite state machine (FSM). [4] Finite state machines explicitly identify the different states a component can be in, and use events as the triggers for transitions from one state to the next (see Fig. 2). Regular expressions on the contrary, focus on the allowed scenarios. With these techniques, the successive states of a component going through a particular sequence of events are only modelled implicitly. From a mathematical point of view, however, both techniques have equal expressive power: every FSM has an equivalent regular expression and vice-versa [19]. Fig. 5 shows the equivalent regular expression for the finite state machine of Fig. 2.

In order to explain contract management for sequence constraints, we need to include some formal definitions of component behaviour. The definitions are inspired by the process algebra of Merode, a presentation of which can be found in [13,43]. The Merode process algebra allows for

[4] Notice that UML uses Harel Statecharts to define this type of constraints. However, the UML action semantics still contain many open ends and proposals for full semantics are tremendously complex [1], far more complex than we need. Additionally, the semantics of parallel composition of state charts is still unclear and does not allow for multiparty interaction (interaction is limited to binary interaction).

```
ORDER = cr_order.(modify_order + cr_orderline + mod_orderline + end_orderline)* .
(ship.invoice. pay .(end_orderline)* .end_order  +  cancel_order)
```

Fig. 5. Equivalent regular expression for the FSM of ORDER.

fixed gate-based multiparty interaction in a similar way as CSP [18] and ACP [4], but is more oriented towards requirements level semantics and makes the type/instance distinction.

In this process algebra, each component is defined as a pair $< \alpha, e >$, with $\alpha \subseteq A$, A being the universe of events, and e being a regular expression over A and such that the events appearing in e are elements of its alphabet α. [5] The regular expression associated with a component defines the set of scenario's that will be accepted by this component. For example, the following sequences of events are examples of scenarios that are accepted by ORDER:

1. cr_order, cancel_order
2. cr_order, mod_order, cr_orderline, cancel_order
3. cr_order, cr_orderline, ship, invoice, pay, end_orderline, end_order
4. cr_order, mod_order, cr_orderline, cr_orderline, mod_orderline, ship, invoice, pay, end_order [6]

Suppose we have two components $C = <\alpha, e>$, and $C' = <\alpha', e'>$. The parallel composition of C and C' is a system the behaviour of which is denoted as $C \,\|\, C'$. When C and C' participate jointly in some events (meaning that $\alpha \cap \alpha' \neq \emptyset$), the system $C \,\|\, C'$ will only accept those scenario's on which both C and C' agree. More formally, assume that $L(C)$ and $L(C')$ denote the sets of scenarios defined by C and C', respectively. Then

$$L(C \,\|\, C') = \{s \in (\alpha \cup \alpha')^* \,|\, s \setminus \alpha \in L(C) \text{ and } s \setminus \alpha' \in C'\}$$

where $(\alpha \cup \alpha')^*$ denotes the set of all possible scenarios containing events from $(\alpha \cup \alpha')$ and $\setminus\alpha$ is the projection operator which drops all events that are not in α from the scenario s.

In a standalone or distributed but tightly integrated environment, the sequence constraints of all object types of the conceptual model are developed in an integrated way, hence reducing the risk of conflicts. In a distributed environment however, sequence constraints are developed in an independent manner by multiple parties. The danger of conflicting sequence constraints is real. Let us reconsider the ordering example in a web service context.

The **supplier** web service offers the possibility to order products. As shown in Fig. 6, the supplier's system keeps track of the products and all orders placed for products. Each order has a link

[5] More detailed definitions are given in Appendix A.

[6] Notice that in order for the system to behave fully correctly, there should be exactly as many end_orderline events as there were cr_orderline events in the life cycle of the order. This constraint can, however, not be enforced solely by the finite state machine. It must be enforced by means of an additional referential integrity constraint specified as precondition for the end_order event in the class ORDER. This issue of so-called "consistent events" is discussed in detail in [29].

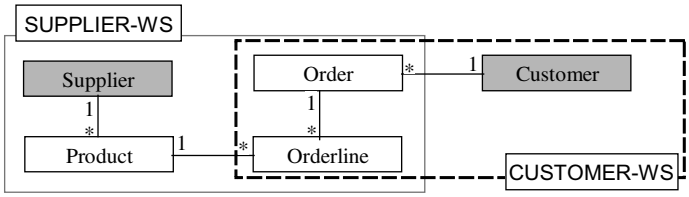

Fig. 6. Distributed components for an order handling system.

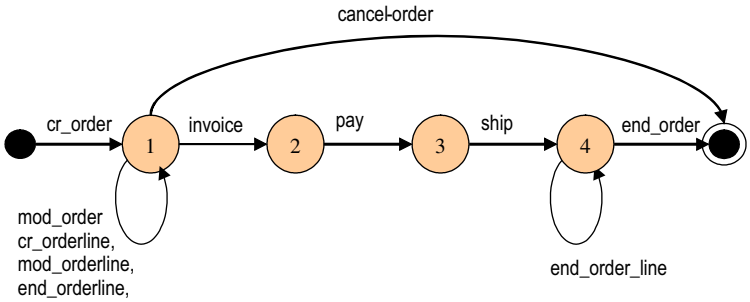

Fig. 7. State machine for ORDER from the supplier's point of view.

to the corresponding **customer**'s web service, but the supplier's system does not manage the customer information. The customer's system keeps track of its own orders. For each order, the customer's system can keep only a stub with a link to the supplier's web service, keep a cache of important attributes of the order or mirror it completely.

Of more importance is the fact that each web service will specify sequence constraints on the events it participates in, according to its local business process. For the sake of simplicity, we assume that both web services use the same names for the events they share [7] and we omit the non-shared events. Let us assume for example that the supplier's business process demands that payment is received before the goods are shipped. The sequence constraints on the order events as specified by the supplier are given in Fig. 7. The customer sticks to the sequence constraints as specified in Fig. 2.

Obviously, this customer will not be able to collaborate successfully with this supplier. The parallel composition of the two finite state machines reveals that the only scenarios that will be jointly accepted by the supplier's business process and by the customer's process are those that end with cancel_order. This can be formalised by means of regular expressions. The behaviour of the customer web service and the supplier web service can be defined as follows with respect to their common events:

Customer_ORDER
$=<\{$cr_order, mod_order, cr_orderline, mod_orderline, end_orderline, ship, invoice, pay,

[7] If this is not the case, a mediation broker could provide a mapping, based on ontological descriptions of the events. This issue is, however, beyond the scope of this paper.

end_orderline, end_order, cancel_order},
cr_order.(mod_order + cr_orderline + mod_orderline + end_orderline)*.
(ship.invoice.pay.(end_orderline)*. end_order + cancel_order)>
Supplier_ORDER
=<{cr_order, mod_order, cr_orderline, mod_orderline, end_orderline, ship, invoice, pay, end_orderline, end_order, cancel_order},
cr_order.(mod_order + cr_orderline + mod_orderline + end_orderline)*.
(invoice.pay.ship.(end_orderline)*. end_order + cancel_order)>

Calculating the parallel composition of these two components yields the following: [8]
(Customer_ORDER) || (Supplier_ORDER)
=<{cr_order, mod_order, cr_orderline, mod_orderline, end_orderline, ship, invoice, pay, end_orderline, end_order, cancel_order},
cr_order.(mod_order + cr_orderline + mod_orderline + end_orderline)*. cancel_order>

Although the resulting regular expression is not empty (which would have indicated a full deadlock), the expression does not contain all the original events: the events *ship*, *invoice* and *pay* are missing in the resulting regular expression. This indicates that shipping, invoicing and payment events will always be rejected by one of the two components because of conflicting sequence constraints. Indeed, the customer will reject the payment event as long as there has been no shipping and invoicing, whereas the supplier will reject the shipping as long as there has been no payment. Obviously, no meaningful collaboration is possible.

The problem of incompatible constraints will be handled differently depending on the systems development context. In a context of fixed partnerships between enterprises, contract analysis can be done at design time. In this kind of situation, conflicting sequence constraints indicate incompatible business processes of the partner companies. The most simple and realistic solution is to resolve the conflicts at the business level by adapting the business processes and the constraints imposed by the components accordingly.

In the context of dynamic web service discovery, incompatible sequence constraints cannot be resolved by modifying the web service and one will continue to search for another web service or set of web services with compatible sequence constraints.

Finally, in a context of dynamic multiparty interaction, where components can (un)subscribe to an event at any time, the calculation of parallel composition can be used by an event dispatcher to manage the subscription policy. The same way of working can be used to manage system evolution in one of the afore-mentioned contexts.

Suppose an event dispatcher ε manages a set of events $\beta = \{b_1, b_2, \ldots, b_n\} \subseteq A$. Initially, when no component has subscribed to ε, the accepted behaviour of ε is any arbitrary sequence of events from β. Formally: $\varepsilon = <\beta, \beta^*>$, with $\beta^* = (b_1 + b_2 + \cdots + b_n)^*$. When a component $C = <\alpha, e>$ subscribes to ε, C will impose restrictions on the acceptable set of scenarios. Since C can be involved in more events than those of β only, we use the projection $C \setminus \beta$ to retain in the definition of C only those sequence constraints that pertain to the events in β. Hence, the new accepted behaviour of ε becomes $<\beta, \beta^*> \| (C \setminus \beta)$, which is equal to $C \setminus \beta$. After accepting the

[8] The algorithm for calculating the parallel composition is given in Appendix A.

subscription of C, the event dispatcher ε will only accept scenarios that are acceptable for C, with respect to the events in β. For every new component D that subscribes to ε, the behaviour of D is incrementally put in parallel with the current behaviour of $\varepsilon : \varepsilon_{new} = \varepsilon_{old} \| (D \setminus \beta)$. If $\varepsilon_{old} \| (D \setminus \beta)$ is different from $D \setminus \beta$, this means that some of the scenarios of D will be refused by the event dispatcher ε. D can decide not to subscribe if $\varepsilon_{old} \| (D \setminus \beta)$ differs too much from $D \setminus \beta$. On the other hand, $\varepsilon_{old} \| (D \setminus \beta)$ might be different from ε_{old} which indicates that scenarios which were accepted before the subscription of D will not be accepted any more. This might be a reason for the event dispatcher ε to refuse the subscription of D as this would reduce the interaction possibilities of the already subscribed components.

Rather than calculating the behaviour of an event dispatcher incrementally as components subscribe to it, it is also possible to define the behaviour of the dispatcher in advance at design time. In such a scheme, a component would then only be able to subscribe with an event dispatcher if its own behaviour definition is not in conflict with the definition of the event dispatcher.

4.2. Distributed versus centralised contract management

In event-based architectures, it is common to implement an event dispatcher in either a centralised or a distributed way [27]. The choice for centralised versus distributed event handling is an architectural choice which has a.o. performance implications. This choice also has a significant impact on the contract management capabilities. Two extremes can be considered: a fully distributed set-up where each event dispatcher takes care of a single event, and a centralised set-up where a single event dispatcher manages all events. In the first case, contract management will be limited to checking preconditions for a single event. In the second case, however, more sophisticated contract management as described in the previous paragraph becomes possible.

From a deadlock detection point of view, a centralised event dispatcher having all events of the universe in its alphabet β is the best solution. Indeed, freedom of deadlock can only be guaranteed if β covers the set of common events for every pair of components subscribed to ε. [9] As illustrated in the next example, the distribution of events over multiple event dispatchers can lead to situations were conflicts are not detected. Suppose for example that in the ordering example, shipping is delegated to a **Courier** web service and that payment will be checked by a **Bank** web service. A *confirm* event is added to signal that an order is ready for further processing. A partial component–event table is given in Table 2.

Let us assume that the part of the behaviour of the four services pertaining to these four events is as follows:

$$\textsc{customer} \setminus \{\text{confirm}, \text{ship}, \text{invoice}, \text{pay}\} = <\{\ldots\}, \text{confirm.ship.invoice.pay}>$$
$$\textsc{supplier} \setminus \{\text{confirm}, \text{ship}, \text{invoice}, \text{pay}\} = <\{\ldots\}, \text{confirm.invoice.pay.ship}>$$
$$\textsc{bank} \setminus \{\text{confirm}, \text{ship}, \text{invoice}, \text{pay}\} = <\{\ldots\}, \text{invoice.pay}>$$
$$\textsc{courier} \setminus \{\text{confirm}, \text{ship}, \text{invoice}, \text{pay}\} = <\{\ldots\}, \text{confirm.ship}>$$

[9] This is a consequence of Theorem 3.4 in [43, p. 44]: $(<\alpha, e> \| <\alpha', e'>) \setminus \beta = (<\alpha, e> \setminus \beta) \| (<\alpha', e'> \setminus \beta)$ only if $\alpha \cap \alpha' \subseteq \beta$. The reverse is not true.

Table 2
Partial component–event table for an order handling system consisting of customer, supplier, courier and bank web services

	Customer	Supplier	Courier	Bank
...				
confirm	M	M	M	
ship	M	M	M	
invoice	M	M		M
pay	M	M		M
...				

Suppose that we have two event dispatchers ε and ε' which handle respectively the events $\beta = \{\text{confirm}, \text{ship}\}$ and $\beta' = \{\text{invoice}, \text{pay}\}$. Customer and Supplier subscribe to both event dispatchers, whereas Courier only subscribes to the first event dispatcher ε, and Bank subscribes to ε'. Contract analysis will not reveal the problem. Indeed:

$$\begin{aligned}
&\varepsilon \,\|\, (\text{customer} \setminus \beta) \,\|\, (\text{supplier} \setminus \beta) \,\|\, (\text{courier} \setminus \beta) \\
&= <\beta, \text{confirm.ship}> \\
&= (\text{customer} \setminus \beta) \\
&= (\text{supplier} \setminus \beta) \\
&= (\text{courier} \setminus \beta)
\end{aligned}$$

Similarly

$$\begin{aligned}
&\varepsilon' \,\|\, (\text{customer} \setminus \beta') \,\|\, (\text{supplier} \setminus \beta') \,\|\, (\text{bank} \setminus \beta') \\
&= <\beta', \text{invoice.pay}> \\
&= (\text{customer} \setminus \beta') \\
&= (\text{supplier} \setminus \beta') \\
&= (\text{bank} \setminus \beta')
\end{aligned}$$

In contrast, an event dispatcher $\varepsilon'' = <\beta'', \beta''^{*}>$ that handles all four events will reveal the problem:

$$\begin{aligned}
&\varepsilon'' \,\|\, (\text{customer} \setminus \beta'') \,\|\, (\text{supplier} \setminus \beta'') \,\|\, (\text{courier} \setminus \beta'') \\
&= <\beta'', 0>
\end{aligned}$$

whereby 0 stands for the deadlock event.

Although a centralised event dispatcher responsible for all known types of events would resolve this problem, it can rapidly become a bottleneck as the number of components it has to serve grows. This becomes even worse when components are distributed over the Internet [8]. In addition, in a situation where components are developed independently and new events are added during the system's lifecycle (such as in a B2B web services context), the development of a centralised event dispatcher is quite impossible. As a result, event dispatching is most likely in itself a distributed activity and the distribution of events across multiple event dispatchers must be carefully analysed in order to maximise the confidence in the overall system. A possible solution

that makes a trade-off between the advantages of a centralised dispatcher and those of a distributed dispatcher is to let a centralised event dispatcher handle the subscription, but to delegate the notification to distributed event handlers. Also, as briefly outlined in Section 3, the concept of existence dependency allows to identify groups of enterprise objects and provides guidelines on how to identify the best partitioning of event types over event dispatchers. This research is still ongoing and beyond the scope of this paper.

5. Architectures for event dispatchers

As existing event-based systems do not offer coordination capabilities, their architecture is mainly guided by the subscription and notification mechanisms. As explained in [27], different implementations can be used to realise a same conceptual event model. In the remainder of this section, we propose different architectures for implementing the coordination aspects. As constraint checking is tightly linked to the notification aspect, each of the proposed architectures implies particular choices for the organisational dimension of the event service [27].

According to the principles of programming by contract [28], preconditions represent an obligation for those components that wish to trigger the execution of an event. As a result, different implementations for the coordination task of an event dispatcher can be defined. Although the possible implementation architectures for the coordination mechanism are a topic of ongoing research, this paper already proposes three broad categories. Far from claiming exhaustiveness, some significant pros and contras are discussed for each.

A straightforward implementation of programming by contract locates precondition checking in the event dispatcher. [10] In this first type of architecture, the event dispatcher needs to know about the preconditions imposed by each application or component on a particular event. Upon invocation of an event by an input service or other component (step 1 in Fig. 8), [11] the event dispatcher first checks whether all preconditions are satisfied by accessing the components through their query interfaces (step 2). If this is the case, the event is dispatched to the participating components by invoking their respective event notification interfaces (step 3). Each of these executes the corresponding method and possibly dispatches the event internally if the application or component is developed in an event-based manner (step 4). Whenever some precondition is not satisfied, the event is not dispatched and the invoking component is notified accordingly.

Although testing the preconditions is in principle the responsibility of the client of a service, one can also develop architectures where the testing of the preconditions resides within the component delivering the service. This can be done in two ways, leading to a second and third type of architecture. In the second type of architecture, the testing of the precondition is separated from the event handling method of the component, meaning that every participant to an event has two methods for each event: one ''check_event'' method to test the preconditions and one ''do_event''

[10] The preconditions include the sequence constraints. The transition function of a state machine is a mapping $(\text{state} \times \text{event}) \rightarrow \text{state}$, which easily allows to implement it for each event as a precondition on a state-variable value.

[11] Notice that each arrow in this figure represents a service invocation which might include a return value being passed to the invoking component. The latter is however implicit and not represented as an additional arrow in the opposite direction.

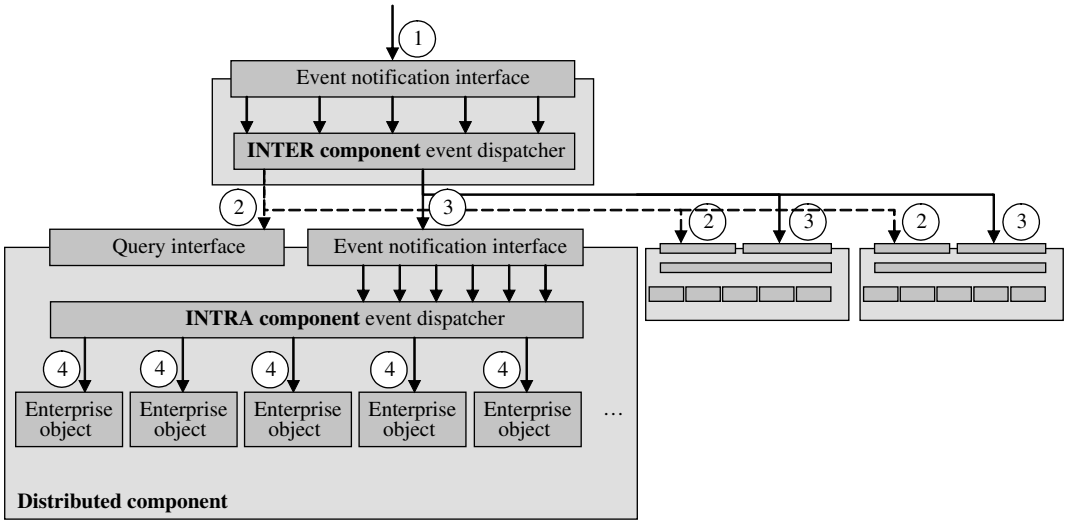

Fig. 8. Centralised precondition checking by the event dispatcher.

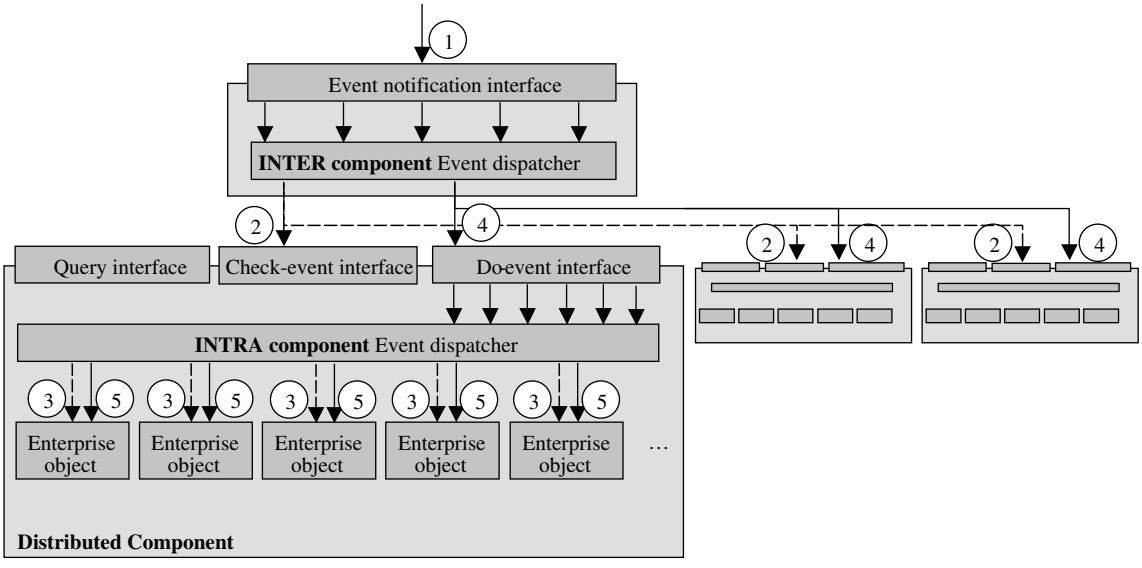

Fig. 9. Distributed precondition checking prior to method-invocation.

method to call the actions required to react to the event. In this architecture (shown in Fig. 9), upon invocation of an event (1), the event dispatcher will first broadcast a "check_event" to all participants (2). Internally, the check_event request can be further delegated to the participating enterprise objects (3). Only if all participants respond positively to the request, the event dispatcher will broadcast the "do_event" request to the participants (4), which is again further dispatched internally (5). If one of the participants responds negatively to the precondition check,

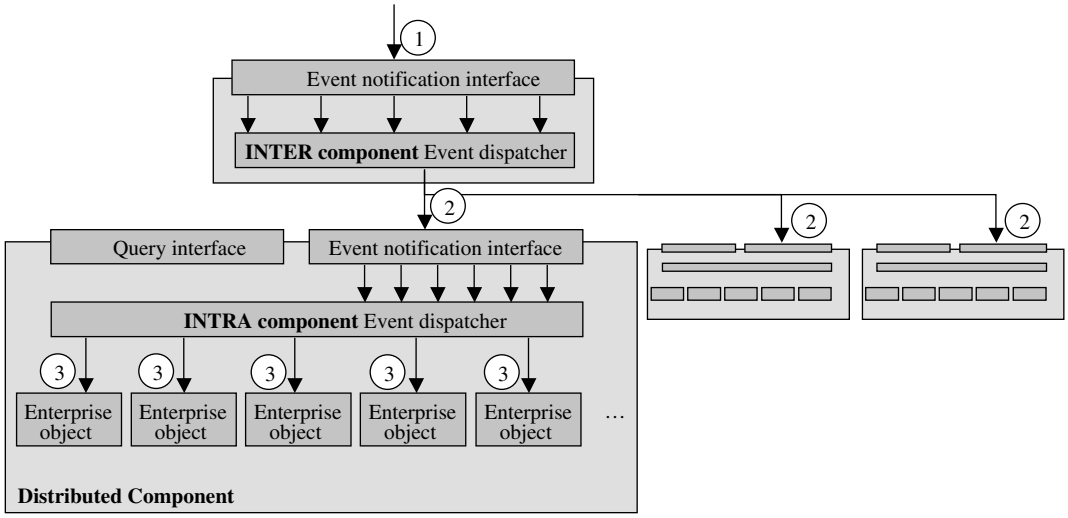

Fig. 10. Distributed event dispatching without prior precondition checking.

the invoking component is notified accordingly. This architecture is somewhat similar to a distributed two-phase commit protocol.

In the third type of architecture (Fig. 10), upon notification of an event (1) the event dispatcher simply broadcasts the event to all participants (2). Each participant checks its local preconditions and broadcasts the event to its local components using the local object–event table (3). In case one of the preconditions is violated, the event dispatcher is notified upon which it will request a rollback of the transaction to all participants.

These three architectures have different strengths and weaknesses with regard to performance and the distribution of components. The first architecture, where precondition checking is located in the event dispatcher is the most efficient one: precondition checking is centralised and the respective components are only notified of events that do not violate any preconditions. In this way, the risk of a costly global rollback is avoided in case some precondition is not met in one of the participating components. The second architecture implies distributed precondition checking, but still avoids the necessity of rollbacks, [12] whereas the third architecture implies distributed precondition checking and will require a rollback if a precondition is not met.

As mentioned before, centralised precondition checking by the event dispatcher will however be difficult to realise in a strongly distributed environment. Especially in a situation where components are developed independently, e.g. in a B2B web services context, the enforcement of preconditions may to a certain extent be encapsulated and hidden from the outside world. For example, a supplier web service may not want to reveal the implementation of the precondition that checks a customer's credit worthiness before shipping an order. As a consequence, the precondition should be checked locally in the supplier web service, not at the level of the central event dispatcher. The latter should only be notified of whether or not the precondition was violated.

[12] At least it avoids the necessity of rollbacks caused by failed events. An underlying transaction management mechanism might still use rollbacks to enforce transaction atomicity at the database level.

Therefore, in a strongly distributed environment, distributed precondition checking is much easier to realise. In that case, there is still the choice between precondition checking prior to method invocation (the second type of architecture) and event dispatching without prior precondition checking (the third type of architecture). The former will require a *locking* mechanism to ensure data consistency in the time interval between the precondition check and the actual processing of the event, which may put a burden on performance. The latter architecture simply tries to resolve precondition violations by inducing a rollback on all participating components *after* they had already processed the event. Whereas a rollback may be a costly operation in terms of performance, the latter may still be the most efficient approach in the case where the number of events for which some precondition is violated is much smaller than the number of "succeeded" events. Furthermore, in more advanced implementations, the components or the event dispatcher may be equipped with the intelligence to induce so-called *compensating actions* if an event partially fails, instead of inducing a costly global rollback. For instance, if the precondition *order quantity* $\leqslant$ *inventory* is violated for a certain order line, the PRODUCT object where the precondition was violated may try to induce a preliminary *refill_stock* event, instead of inducing a rollback of the entire transaction.

Regardless the actual architecture, it is of utter importance that it is not limited to *event notification*, but that the *atomicity* of an event is enforced among all participating components. For that purpose a transaction management mechanism is to be used underneath, e.g. in a web services context this could be WS-Transaction [7] or BTP [9]. The transaction management mechanism enforces the all-or-nothing aspect of events. The latter is key to the coordination mechanism presented in this paper: the participating components are not only notified as in traditional event-based systems [8,12,17,21,27], but each event is indeed an atomic unit of work that is either accepted and processed in all components, either it is rejected and no action is taken in any component at all. All participating components may enforce constraints that together determine the composite system's behaviour.

6. Related work

The business event-based coordination model presented in this paper (or BECO model for short) spans different areas of research. As we concentrate on the modelling and design level, the BECO model is related to event-based requirements engineering methods. Also, the event-based paradigm has become an emerging architectural concept for the realisation of many different types of systems [3,12,27,38]. This has resulted in the definition of many different types of event generation, observation and notification systems. Furthermore, event-based coordination can be formalised by process algebras supporting multiparty interaction. The next paragraphs review related research in these three areas.

6.1. Event-based conceptual modelling

Although the majority of event-based systems are in the area of architectures for distributed systems, a few requirements engineering methods can be characterised as event-based as well. The BECO model is mainly influenced by the ideas of Merode. Although Merode is and object-oriented development method, it recognises events as independent concepts, not subordinated to objects. But

also OOSSADM [36], Syntropy [11], Catalysis [15] and Remora [37] are examples of requirements engineering or analysis and design methods with a strong emphasis on events. OOSSADM, Syntropy and Catalysis deal with events in a rather informal way. Remora on the contrary, deals very explicitly with events and is supported by a formal model. In Remora, a c-event represents the *class* of events that ascertain only one type of state change of objects corresponding to the same c-object. As a result, a c-event belongs to exactly one c-object (= class). As a result Remora allows for a one-to-many communication paradigm, but not explicitly for many-to-many communication. This is a major difference between the BECO model and Remora: in the BECO model an event type (∼c-event in Remora) is not attached to a single object type (∼c-object in Remora); an event type is defined as an atomic unit of behaviour that can be *invoked* by *many* parties and that may require the coordinated *participation* of *many* objects.

6.2. Design and architectures of notification services

Beyond the area of requirements modelling, the event-based architecture is a basic architectural pattern that has been used for the realisation of a wide range of systems [40]. The basic idea of an event-based architecture is a structure of event generation, observation and notification: on the one hand there are events that are produced by producing entities and on the other hand there are consuming entities that wish to be notified of the occurrence of the events they are interested in. The event architecture defines the ways to produce or observe events and to notify consumers of event occurrences. The notification is usually the responsibility of a connector (called an event service, an event dispatcher or a bus) and it is hidden to the component that generated the event [8]. The notion of event captures the concept of an autonomous asynchronous occurrence. Some architectures also allow consumers to be notified of composed events or of event patterns rather than of single events only [12]. Additionally, events can be broadly categorised as operation-triggered, explicitly-posted and timer events [31].

For event-based programming systems a distinction can be made between the event-model and the event service dimension, the first presenting the component view and the latter defining the implementation paradigm [27]. Likewise, the MDA of the OMG [33] makes a difference between platform independent models (PIMs) and platform specific models (PSMs). The BECO model is an example of an implicit event model as consuming entities subscribe to an event type rather than at another entity or at a particular mediator [27]. The architectures presented in Section 4 are examples of distributed-collocated event services with multiple centralised intermediates [27].

Despite the many similarities with existing event-based architectures, there are some substantial differences as well. Many event-based architectures represent a *one* to many communication paradigm: an event is produced by or observed from *one* entity and notified to *many* interested parties. When an event is defined as a property of *one* object (e.g. a change in attribute or a tick of a clock/timer-object) it can only be produced by this one component. [13] As a result, it is sent from *one* object

[13] Rosenblum [38] defines an event as "the instantaneous effect of the (normal or abnormal) termination of an invocation of an operation on a object and it occurs at that object's location". In Remora [37] an event ascertains a state change of one object. Fiege [17] defines an event as "any transient occurrence of a happening of interest, i.e., a state change in some component. The affected component issues a notification that describes the event." Yeast [21] defines a primitive event "as a change in the value of an attribute of an object belonging to some class". In CORBA [34], an event is simply a message that one object communicates to another.

to many interested parties. In contrast, the BECO model is a many to many communication paradigm: an event can be *invoked* by multiple parties; an event is not the result of the observation of a state change in one object.

However, the *major difference* between the BECO model and existing event-based architectures lies in the *aspect of coordination*. In these existing approaches, when multiple consumers are notified of an event occurrence, it is assumed that these consumers will act *independently* from each other and that no response to the event producer is required. The presentation of JEDI in [12] mentions the issue of the need for a return value in some cases, but no actual solution is presented. In the BECO model, the invoking component can receive feedback on the execution status of the coordinated action associated with the triggered event. Moreover, in addition to the possibility of providing the event producer with a return value, the responses of the event consumers need to be coordinated (either *all* consumers execute the corresponding action successfully or all the action associated to the event are rolled back). Likewise, the event-based coordination architecture allows event consumers to impose constraints on the invocation of events. This feature is not present in the afore-mentioned event-based architectures.

Another major difference with existing event-based architectures is that in the BECO model a distinction is made between the class and its instances. In many event-based architectures, individual object *instances* will subscribe to events. In contrast, in event-based coordination subscription to event types is at class level but notification is at instance level. More particularly, although it is a *class* that subscribes to an event type, an event occurrence will be notified to a *single* occurrence of the subscribed consumer class, rather than to all occurrences of the consumer class (e.g. a ship event will be notified to *one particular* customer service). As a result, the notifying instance should provide the necessary information for identifying the required consumer instances. Hence, the interaction cannot be anonymous [27].

Finally, a difference can be noted on the level at which the concept "event" is considered. In most event-based architectures, an event can be defined as an *information system* event, such as for example the update of a file or the tick of a clock. In our approach, we only consider *business* events. Similarly, we only consider interaction between software components whereas in Yeast [21] for example, event notifications are delivered to human beings.

6.3. Process algebras for event systems: multiparty interaction and distributed systems

Although the event-based architecture has been used to implement a variety of systems, to our knowledge, none of them specify the semantics of event notification by means of a formal model (e.g. by means of process algebra). The formal semantics of the BECO model are defined by means of a CSP-like process algebra. Within the framework proposed by Joung [20], our process algebra can best be characterized as gate-based multiparty interaction, with conjunctive parallelism. In the current formalisation the set of participants is fixed, but it is our intent to research the possibility to have a variable set of participants by formalising the semantics of the subscription mechanism as well.

A major difference between the process algebra for the BECO model and existing process algebras stem from the fact that the BECO model follows the major principles of the object-oriented model and hence makes a distinction between the type level and the instance level, uses identifier addressing, and encapsulates data and behaviour into the class definition. Most process

algebras on the contrary are instance level models, [14] use channel addressing and separate data from behaviour.

As a result of these differences, many problems and solutions of existing process algebras have to be reconsidered in the context of the BECO model. As an example, because of the Class/Instance distinction and the possibility of creating and destroying instances in a class, the issues of deadlock and starvation are very different than in a process algebra such as CSP. [15] The BECO model addresses part of the problem of deadlock by verifying the components for possible contradicting constraints before-hand (= at subscription time). The fairness and starvation problem is more difficult to handle as we cannot influence the timing of event invocation. In a first time, fairness and starvation will be beyond the scope of our research.

More recently, the event-paradigm has also been used in a Web-service context [23,31]. This research focuses on operation-triggered events which are posted before or after the invocation of a specific operation. BECO focuses on explicitly-posted events, leading to the coordinated invocation of multiple operations. BECO also differs from the Event–Condition–Action framework [23] in that it is the posting of the event that is subject to constraints and not the triggered action.

7. Conclusions and further research

This paper presented an event-based interaction mechanism between components and/or applications, with events playing the role of *contracts*, into which each participating component is able to insert its own preconditions. This mechanism allows the coordination of simultaneous actions in these components, to which the events are *broadcast in parallel*. Only if preconditions are satisfied in all components, each component responds to the event by executing a method that updates its state. In this way, events trigger a *distributed transaction* over the participating components. As illustrated in Fig. 4, this mechanism can be applied recursively, with a complex component propagating an event notification to its own constituent components. A component's preconditions to an event consist of the combined preconditions in its own components and its reaction to the event consists of the combined method executions in its respective components.

Since multiple parties are allowed to impose constraints on the same event by means of preconditions, the preconditions should be investigated for contradictions and inconsistencies. This kind of verification requires a formal approach to requirements and specification. Although UML has become a de facto standard notation for specifications, it lacks precise and formal semantics. The need for formal underpinning of UML has long been recognised and significant advances have been made [16,22,35,39]. Many of these efforts are, however, limited to the isolated definition of a single modelling notation [5,16]. Although some advances have been made towards a more formal underpinning of the behavioural aspects of a system specified with UML [6,10,24], none of the approaches is currently able to calculate and analyse the *system* behaviour, starting from the

[14] In an instance model, the description of the Dining Philosophers counts exactly five philosophers and five forks [14]. A class-level model would define this as a system of two collaborating classes "philosopher" and "fork" of which instances can dynamically be added to and removed from the system.

[15] In the class-level version of the Dining Philosophers, a deadlock situation of five philosophers holding each one fork can be resolved by creating a sixth fork.

behavioural descriptions of the individual components. It is exactly to achieve this type of verification of global system behaviour that Merode has introduced the notion of events. In its current incarnation, Merode already allows for verifying contracts with respect to those constraints that relate to sequencing of events [13]. At the component level, the allowed sequences of business events can be documented by means of a finite state machine. This allows calculating the overall system behaviour and identifying problems such as deadlock and starvation. In this way, it becomes possible to check whether a set of components is able to cooperate, i.e. whether a scenario of consecutive events exists that results in a successful conclusion of a business transaction. The latter is also applicable to *service discovery*, i.e. to search for services that are compatible from a business process point of view. Further research will investigate how other types of preconditions (such as e.g. domain constraints on attributes or parameter values) can be included in the verification of the contracts.

A current limitation is that the event sequence constraints only represent *control dependencies* between the component's activities. There is no direct construct for modelling *data dependencies*. An example of the latter could be the requirement that an *invoice* event is only allowed after some *total_price* attribute has been assigned a value. In the current incarnation, data dependencies are to be modelled indirectly, through sequence constraints that enforce events that assign a value to a certain attribute to occur before events that require inspection of this attribute as part of their processing. Note that this issue has not been completely resolved in languages such as BPEL4WS [46] either.

In addition to its contracting aspect, the event-based broadcasting paradigm offers the additional advantage of a complete separation of the coordination aspects from the functionality aspects. The responsibility for the coordination resides completely within the event dispatcher, whereas the functionality part resides within the participating applications or components.

Although a number of implementation frameworks exist for event observation and notification [3,12,17,21,27], most of these do not completely suit the needs of the event-based coordination paradigm described in this paper. The event model of these frameworks mostly associates events with a single invoker. Additionally, these types of frameworks are not concerned with the semantics of events nor with their potential role as contract between components. As a result, ongoing research focuses on the ways to implement an event-based coordination mechanism and an event dispatcher for different types of environments, e.g. standalone or distributed, tightly coupled or loosely coupled etc. This paper already presented three basic types of architectures. An example of a fully operational standalone implementation as the result of an application integration project is discussed in [25]. A prototype architecture also exists for the particular situation of an inter-enterprise web services environment [26].

The latter publication also reveals another apparent advantage of the event-based approach, which was not emphasised in this paper: the fact that event propagation is a *dispatching* mechanism offers substantial advantages in terms of choreography definition and transaction management. Indeed, it is a many-to-many interaction mechanism, which allows modelling a single business event as resulting in multiple simultaneous updates in multiple components. The latter is much easier to describe and coordinate than the myriad of message exchanges that could make out a single business transaction in a pure one-to-one method invocation approach. The specification of a business process is only concerned with the appropriate consecution of events, not with the more complex consecution of underlying message exchanges that take care of event *notification*.

Moreover, with an event-based approach, a firm distinction can be made between operations that "read from" attributes in another application or component and operations that "write to" attributes in another application or component. These types of operations are intertwined when a generic method invocation approach is used. However, in an event-based approach, the clear distinction between *attribute inspections* (which only "read from" attributes and do not entail state changes) and *business events* (which "write to" attributes and do entail state changes) allows for focusing on only the latter with respect to choreography and transaction management.

A last topic of ongoing and future research, especially relevant in a web services context, is the concept of dynamic service discovery and invocation. For that purpose, a web service description should not be restricted to its WSDL [45] interface definition, i.e. which methods can be called, but should also reveal the sequence in which these methods have to be called to successfully accomplish a business transaction. Multiple languages already exist for that purpose, e.g. XLang [44], BPEL4WS [46], BPML [2], etc., but the event-based approach stands out in that it allows verifying whether a successful scenario exists over a group of services, each with its own preconditions. As already discussed in Section 5, a further enhancement is to allow for negotiation, alternative scenarios and compensating actions if an event fails because of a precondition violation in one of the participating services. Again, in comparison to a pure one-to-one interaction mechanism, the concept of event-based coordination and events being conceived as contracts between multiple parties appears favourable in the development of algorithms for mediation and compensating actions.

Appendix A. Formal definition of regular expressions, regular languages and parallel composition

Let A be the universe of business event types. Two special event types are defined: the "do nothing" event type denoted by the symbol 1 and the deadlock event type denoted by the symbol 0. Regular expressions over A are built by means of the operators '+' (choice), '$\cdot$' (sequence) and '$*$' (iteration). $R(A)$ is the set of regular expressions over A and is defined as follows:

$$0, 1 \in R(A)$$
$$\forall a \in A : a \in R(A)$$
$$\forall e,\ e' \in R(A) : e + e' \in R(A), e \cdot e' \in R(A),\ (e)^* \in R(A)$$

The operators satisfy the following laws:

(1) $0 + e = e$,
(2) $e + e = e$,
(3) $e + e' = e' + e$,
(4) $e + (e' + e'') = (e + e') + e''$,
(5) $1 \cdot e = e = e \cdot 1$,
(6) $0 \cdot e = 0 = e \cdot 0$,
(7) $e \cdot (e' \cdot e'') = (e \cdot e') \cdot e''$,
(8) $e \cdot (e' + e'') = e \cdot e' + e \cdot e''$,
(9) $(e + e') \cdot e'' = e \cdot e'' + e' \cdot e''$,
(10) $e^* = 1 + e + e \cdot e + e \cdot e \cdot e + e \cdot e \cdot e \cdot e + \cdots$.

Laws (6) and (8) are somewhat unusual compared to other process algebras such as CSP [18], CCS [30] and ACP [4]. As to law (6), a process $0 \cdot e$ is a process that first is in deadlock and then behaves as e. Obviously, deadlock is an absorbing state for a process and thus can never be followed by any action. Therefore $0 \cdot e = 0$. The expression $e \cdot 0$ describes a process that first behaves like e and then deadlocks. At least this process can undertake some meaningful actions and therefore most process algebras define that $e \cdot 0 \neq 0$. This way of working is for example mandatory when the process algebra is used to study the behaviour of an implementation. However, when specifying business processes, we are only interested in correct behaviour, this is, processes and systems that behave correctly up to the end of their lives and never deadlock. At the business specification level, $e \cdot 0$ is just as "bad" as 0 and therefore in Merode $e \cdot 0 = 0$. A more thorough motivation of the laws (6) and (8) can be found in [13,43].

A scenario over A is a sequence of events of A. The set of all possible scenarios over A is denoted as A^*. The set of scenarios defined by a regular expression e from $R(A)$ is called the regular language defined by e and is denoted $L(e)$.

A^* is the set of scenarios over A and is recursively defined as follows:

$1 \in A^*$
$\forall a \in A : a \in A^*$
Let $s, t \in A^*$, then $s \wedge t \in A^*$

Catenation '$\wedge$' satisfies the following properties:

$\forall s \in A^* : 1 \wedge s = s = s \wedge 1$
$\forall s, t, u \in A : s \wedge (t \wedge u) = (s \wedge t) \wedge u$

The regular language defined by the regular expression is defined by

$L(0) = \emptyset$
$L(1) = \{1\}$
$\forall a \in E : L(a) = \{a\}$

$\forall e,\ e' \in R(E)$:

$L(e \cdot e') = L(e) \cdot L(e')$
$L(e + e') = L(e) \cup L(e')$
$L(e^*) = L(e)^*$

where $L(e) \cdot L(e') = \{s \wedge t \,|\, s \in L(e) \text{ and } t \in L(e')\}$ and $L(e^*) = \{1\} \cup L(e) \cup L(e) \cdot L(e) \cup L(e) \cdot L(e) \cdot L(e) \cup \ldots$

The projection operator $\backslash\beta$ with $\beta \subseteq A$ replaces events that are not in β with 1 both in individual scenarios and regular expressions:

$0 \setminus \beta = 0$
$1 \setminus \beta = 1$

$\forall a \in A : (a \in \beta \Longleftrightarrow a \setminus \beta = a)$ and $(a \notin \beta \Longleftrightarrow a \setminus \beta = 1)$
$\forall s, t \in A^* : (s \wedge t) \setminus \beta = (s \setminus \beta) \wedge (t \setminus \beta)$
$\forall e,\ e' \in R(A) : (e + e') \setminus \beta = (e \setminus \beta) + (e' \setminus \beta), (e \cdot e') \setminus \beta = (e \setminus \beta).(e' \setminus \beta), (e^*) \setminus \beta = (e \setminus \beta)^*$

The parallel composition operator $\|$ lets object types synchronise on common event types. An object type (or component) is defined as a tuple $<\alpha, e>$, with $\alpha \subseteq A$ and $e \in R(A)$, such that $e \setminus \alpha = e$.

Let $<\alpha, e>$ and $<\alpha', e'>$ be two object types then

$$<\alpha, e> \ \| \ <\alpha', e'> = <\alpha \cup \alpha', e''> \quad \text{with } e'' \in R(A) \text{ such that}$$
$$L(e'') = \{s \in (\alpha \cup \alpha')^* \,\|\, s \setminus \alpha \in L(e) \text{ and } s \setminus \alpha' \in L(e')\}$$

The calculation of the parallel composition of two regular expression is done by means of finite state machines. The algorithms for transforming a regular expression into an equivalent finite state machine and for calculating the result of a sum, sequence and iteration can be found in any book on formal languages and automata theory, e.g. [19]. The algorithms for the parallel operator can be derived from the algorithm for calculating the intersection of languages and is given below.

An FSM is defined as a tuple $(\Sigma, Q, \Delta, q, F)$, whereby

- Σ is the set of tokens in the input language of the automaton (in our case this is the set of events or alphabet of the FSM),
- Q is the set of states of the FSM,
- Δ is the transition function from $Q \times S$ to Q,
- q is the initial state, and
- F is the set of final states.

Given two deterministic and completely specified FSMs $M_1 = (\Sigma_1, Q_1, \Delta_1, q_1, F_1)$ and $M_2 = (\Sigma_2, Q_2, \Delta_2, q_2, F_2)$, such that the language accepted by M_1 is e and the language accepted by M_2 is e'. A new FSM $M = M_1 \| M_2$, the language of which is $e \| e'$ is defined as follows:

$M = (S_1 \cup S_2, Q_1 \times Q_2, \Delta, (q_1, q_2), F_1 \times F_2)$ such that
$\forall a \in S_1 \cap S_2$:
if $\Delta_1(q_1, a) = q_1'$ and $\Delta_2(q_2, a) = q_2'$
then $\Delta((q_1, q_2), a) = \{(q_1', q_2')\}$
$\forall a \in S_1 \setminus S_2$:
if $\Delta_1(q_1, a) = q_1'$
then $\Delta((q_1, q_2), a) = \{(q_1', q_2)\}$
$\forall a \in S_2 \setminus S_1$:
if $\Delta_2(q_2, a) = q_2'$
then $\Delta((q_1, q_2), a) = \{(q_1, q_2')\}$

References

[1] Action Semantics Consortium, Action Semantics for the UML, Available from <http://www.kc.com/as_site/home.html>, accessed on 14/11/2003.

[2] A. Arkin, Business Process Modeling Language, BPMI draft specification, 2002, Available from <BPMI.org> accessed 11/2000.

[3] J. Bacon, J. Bates, R. Hayton, K. Moody, Using events to build distributed applications, in: Proceedings of the Second International Workshop in Services in Distributed Networked Environments, IEEE Computer Society Press, Silver Spring, MD, June 1995, pp. 148–155.

[4] J.C.M. Baeten, Procesalgebra, Kluwer programmatuurkunde, Deventer, 1986.

[5] R.H. Bourdeau, B.H.C. Cheng, A formal semantics for object model diagrams, IEEE Transactions on Software Engineering 21 (10) (1995) 799–821.

[6] J.M. Bruel, J. Lilius, A. Moreira, R.B. France, Defining precise semantics for UML, in: ECOOP 2000, Workshop Reader, Lecture Notes in Computer Science, vol. 1964, Springer, Berlin, 2000, pp. 113–122.

[7] Cabrera, G. Copeland, B. Cox T. Freund, J. Klein, T. Storey, S. Thatte, Specification: Web Services Transaction (WS-Transaction), 2002, Available from <http://www-106.ibm.com/developerworks/webservices/library/ws-tran-spec/>, accessed on 22/11/2003.

[8] A. Carzaniga, E. Di Nitto, D.S. Rosenblum, A.L. Wolf, Issues in supporting event-based architectural styles, in: Proceedings of the Third International Software Architecture Workshop, ACM Press, New York, November 1998, pp. 17–20.

[9] A. Ceponkus, P. Furniss, A. Green, Business Transaction Protocol, OASIS Committee Specification, 2001, Available from <http://xml.coverpage.org/OASIS-BTP-Specification-DRAFT090.pdf>, accessed on 2/12/2003.

[10] K.S. Cheung, K.O. Chow, T.Y. Cheung, Consistency analysis on lifecycle model and interaction model, in: Proceedings of the International Conference on Object Oriented Information Systems, 9–11 September, Paris, Springer, Berlin, 1998, pp. 427–441.

[11] S. Cook, J. Daniels, Syntropy, Prentice Hall, Englewood Cliffs, NJ, 1994.

[12] G. Cugola, E. Di Nitto, A. Fuggetta, The JEDI event-based infrastructure and its application to the development of the OPSS WFMS, IEEE Transactions on Software Engineering 27 (9) (2001) 827–850.

[13] G. Dedene, M. Snoeck, Formal deadlock elimination in an object-oriented conceptual schema, Data and Knowledge Engineering 15 (1) (1995) 1–30.

[14] E. Dijkstra, Cooperating sequential processes, in: F. Genuys (Ed.), Programming Languages, Academic Press, New York, 1968, pp. 43–112.

[15] D. D'Souza, A.C. Wils, Objects, Components, and Frameworks with UML, the Catalysis approach, Addison-Wesley, Reading, MA, 1999.

[16] A. Evans, R. France, K. Lano, B. Rumpe, Developing the UML as a formal modelling notation, in: J. Bézivin, P.-A. Muller (Eds.), UML'98 Beyond the Notation, International Workshop, Mulhouse France, Lecture Notes in Computer Science, vol. 1618, Springer, Berlin, 1998, pp. 397–407.

[17] L. Fiege, M. Mezini, G. Mühl, A. Buchmann, Engineering event-based systems with scopes, in: Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02), Lecture Notes in Computer Science, vol. 2374, Springer, Berlin, 2002, pp. 309–334.

[18] C.A.R. Hoare, Communicating Sequential Processes, in: Prentice-Hall International, Series in Computer Science, 1985.

[19] J.E. Hopcroft, J.D. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, Reading, MA, 1969.

[20] Y. Joung, S.A. Smolka, A comprehensive study of the complexity of multiparty interaction, Journal of the ACM 43 (1) (1996) 75–115.

[21] B. Krishnamurthy, D.S. Rosenblum, Yeast: a general purpose event-action system, IEEE Transactions on Software Engineering 21 (10) (1995) 845–857.

[22] L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar, Workshop on Consistency Problems in UML-based software development, Workshop Materials,Research Report 2002:06, Blekinge Institute of Technology, Ronneby 2002,

Workshop at the UML 2002 Conference, Available from <http://www.ipd.bth.se/consistencyUML/>, accessed on 11/2003.
[23] H. Lam, S.Y.W. Su, Component interoperability in a virtual enterprise using events/triggers/rules, in: Proceedings of OOPSLA '98 Workshop on Objects, Components, and Virtual Enterprise, October 18–22, 1998, Vancouver, BC, Canada, 1998, pp. 47–53.
[24] A. Le Grand, Specialisation of object lifecycles, in: Proceedings of the International Conference on Object Oriented Information Systems, 9–11 September, Paris, Springer, Berlin, 1998, pp. 259–275.
[25] W. Lemahieu, M. Snoeck, C. Michiels, An enterprise layer based approach to application service integration, Business Process Management Journal 9 (6) (2003) 760–775 (Special issue on Process outsourcing & application service provision).
[26] W. Lemahieu, M. Snoeck, C. Michiels, F. Goethals, G. Dedene, J. Vandenbulcke, Event-based web service description and coordination, in: Web Services, e-Business, and the Semantic Web Workshop of the CAiSE'03 Conference, Lecture Notes in Computer Science Series, Springer, Berlin (accepted).
[27] R. Meier, V. Cahill, Taxonomy of distributed event-based programming systems, Technical Report, TCD-CS-2002, Department of Computer Science, Trinity College Dublin, Ireland, March 2002.
[28] B. Meyer, Object-oriented software construction, second ed., Prentice-Hall, Englewood Cliffs, NJ, 1997.
[29] C. Michiels, M. Snoeck, W. Lemahieu, F. Goethals, G. Dedene, A layered architecture sustaining model driven and event driven software development, in: Proceedings of the Perspectives of System Informatics, 9–12 July 2003, Novosibirsk, Akademgorodok, Russia, Springer Lecture Notes in Computer Science, 2890 (2003). Springer, Berlin, pp. 58–65.
[30] R. Milner, A calculus of communicating systems, in: Lecture Notes in Computer Science, Springer, Berlin, 1980.
[31] K. Nagarajan, H. Lam, S.Y.W. Su, Integration of business event and rule management with the web service model, International Journal of Web Services Research 1 (1) (2004) 41–57.
[32] OMG, Unified Modelling Language, Available from <www.omg.org/uml>, accessed 11/2003.
[33] OMG, Model-Driven Architecture, Available from <http://www.omg.org/mda>, accessed 22/11/2003.
[34] OMG, CORBA, Available from <http://www.omg.org/corba>, accessed 22/11/2003.
[35] pUML, The precise UML group, Available from <http://www.cs.york.ac.uk/puml/>, accessed 11/2003.
[36] K. Robinson, G. Berrisford, Object-oriented SSADM, Wiley, Chichester, 1994.
[37] C. Rolland, Chr. Richard, The REMORA methodology for information system design and management, in: T.W. Olle, G.H. Sol, A.A. Verrijn-Stuart (Eds.), Information Systems Design Methodologies: A Comparative Review, IFIP, North Holland, 1982, pp. 369–426.
[38] D.S. Rosenblum, A.L. Wolf, A design framework for internet-scale event observation and notification, in: M. Jazayeri, H. Schauer (Eds.), Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), Lecture Notes in Computer Science, vol. 1013, Springer Verlag, Berlin, 1997, pp. 344–360.
[39] B. Rumpe, A note on semantics (with an emphasis on UML), in: Second ECOOP Workshop on Precise Behavioural Semantics, in: H. Kilov, B. Rumpe (Eds.), Technische Universität München, TUM-I9813, 1998.
[40] M. Shaw, D. Garlan, Software architecture: perspectives on an emerging discipline, Prentice-Hall, Upper Saddle River, NJ, 1996.
[41] M. Snoeck, G. Dedene, Existence dependency: The key to semantic integrity between structural and behavioral aspects of object types, IEEE Transactions on Software Engineering 24 (4) (1998) 233–251.
[42] M. Snoeck, G. Dedene, M. Verhelst, A. Depuydt, 1999, Object-oriented enterprise modelling with MERODE, Leuvense Universitaire Pers, Leuven, 1999.
[43] M. Snoeck, On a process algebra approach for the construction and analysis of MERODE-based conceptual models, Ph.D. Dissertation, K.U. Leuven, Faculty of Sciences and Faculty of Applied Economic Sciences, Department of Computer Science, 1995.
[44] S. Thatte, XLANG Web services for business process design, Available from <http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm>, accessed 11/2003.
[45] Web Services Description Language (WSDL) 1.0 specification, Available from <http://msdn.microsoft.com/xml/general/wsdl.asp>, 2001, accessed 11/2003.
[46] S. Weerawana, F. Curbera, Business Process with BPEL4WS, IBM white paper (2002), Available from <http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1/>, accessed 22/11/2003.

Monique Snoeck obtained her Ph.D. in May 1995 from the Department of Computer Science of the Katholieke Universiteit Leuven with a thesis that lays the formal foundations of the object-oriented business modelling method MERODE. Since then she has done further research in the area of formal methods for object-oriented conceptual modelling. She now is Associate Professor with the Management Information Systems Group of the Department of Applied Economic Sciences at the Katholieke Universiteit Leuven in Belgium. She has been involved in several industrial conceptual modelling projects. Her research interest are object oriented conceptual modelling, software architecture and software quality.

Wilfried Lemahieu holds a Ph.D. from the Department of Applied Economic Sciences of the Katholieke Universiteit Leuven, Belgium (1999). At present, he is associate professor at the Management Informatics research group of this department. His teaching includes Database Management, Data Storage Architectures and Management Informatics. His research interests comprise distributed object architectures and web services, object-relational and object-oriented database systems and hypermedia systems.

Frank Goethals completed his Master studies in economics (option informatics), at the K.U. Leuven, Belgium, in 2001. He is presently researching for a Ph.D. under the theme of 'Extended Enterprise Infrastructures'. This research is conducted at the K.U. Leuven under the guidance of professor J. Vandenbulcke, and is financed by SAP Belgium. He has a strong interest in Web services technology and Enterprise Architecture, and is investigating how supply chain partners could integrate their systems efficiently and effectively.

Guido Dedene studied Mathematics and Physics at the Catholic University of Leuven (K.U. Leuven) and obtained a National Science Grant Founded Ph.D. on General Relativity Theory. He was Systems Analyst, Systems Engineer and Management Assistant at a major Belgian Computer Center. In 1986 he returned to K.U. Leuven at the Faculty of Economics and Applied Economics, where he became full professor in 1997. His teaching and research activities focus on consistent formal development of Business-oriented Information Systems as well as the Management Aspects of such Systems. He also holds a Chair on the ''Development of Information and Communication Systems'' at the University of Amsterdam, and teaches in several Management-oriented programs, including Vlerick Leuven Gent School of Management. He is publishing on a regular basis in International Refereed Journals and Conferences. He obtained the ''Software Best Practice Award'' on behalf of IEEE Software. He is active in several International Development Programs.

Jacques A. Vandenbulcke is professor at the Department of Applied Economic Sciences of the Catholic University of Leuven, Belgium. His main research interests are in Database management, Data modelling, and Business Information Systems. He is co-ordinator of the Leuven Institute for Research on Information Systems (LIRIS) and holder of the SAP-chair on 'Extended enterprise ICT-infrastructures'. He is president of 'Studiecentrum voor Automatische Informatieverwerking (SAI)', the largest society for computer professionals in Belgium, and co-founder and vice-president of the 'Production and Inventory Control Society (PICS)' in Belgium.