The BROKER/SERVICES MODEL for the Design of Cooperative Process-Oriented Environments

Dimitrios Tombros, Andreas Geppert, Klaus R. Dittrich

Technical Report 97.06
June 1997 (Revised November 1997)

Department of Computer Science, University of Zurich
Winterthurerstr. 190, CH-8057 Zurich, Switzerland
{tombros | geppert | dittrich}@ifi.unizh.ch

Abstract

Cooperative process-oriented environments (CPEs) are typically large-scale distributed heterogeneous systems whose behavior is defined by processes. A CPE has to provide mechanisms for the integration and the coordination of applications of considerably different technologies running on heterogeneous platforms, such as diverse database management systems. In this paper, we propose the BROKER/SERVICES MODEL as a software architecture model for CPEs. In this model, actors in a CPE (software systems or humans) are specified and implemented as reactive components. This model is also capable of modeling the functionality of CPE-components as well as component interaction in a process-independent as well as a process-specific manner. In particular, the model can be applied to integrate database and non-database applications as well as manual tasks into architecturally coherent cooperative information systems.

Keywords: construction of cooperative information systems, event-driven workflow management

1 Introduction and Motivation

The functionality of complex information systems is often achieved through the cooperation of collections of distributed (heterogeneous) software systems and humans. In many cases, the ways tasks of a system are accomplished is defined (explicitly or implicitly) in terms of *processes*. This kind of information system is thus referred to as *cooperative process-oriented environment* (CPE); typical examples are workflow systems [10, 13] and process-centered software development environments. Building a CPE is a complex task that has to meet several requirements. Concretely, CPE-construction should support

- the integration of existing heterogeneous applications,
- the definition of the overall CPE-behavior in terms of processes,
- distribution, and
- maintenance of the autonomy of component systems.

Information systems in general and CPE in particular are usually built by integrating existing operational component systems. These are often heterogeneous with respect to platforms they run on (hardware, operating systems), storage facilities they use (i.e., different database systems or files), manipulated data formats, and the external interfaces they provide. Thus, a crucial step during the construction of a CPE is to integrate these heterogeneous component systems into a coherent whole based on a common, global system model.

In order to define how the functionality of components contributes to the tasks of the CPE, their behavior must be expressed in some global notation. Thus, *control integration* (or *task integration*) is important. Tasks in the CPE should be definable and enactable as processes using the component systems' functionality. Integration should however, preserve the autonomy of components, which may still operate independently of the CPE. Furthermore, component systems typically will reside on different although connected physical locations, so that the CPE must be able to execute processes in a distributed environment. The resulting CPE-architecture must be able to flexibly accommodate changes in the component system functionality and structure as well as in the process definitions without requiring a redesign of the whole CPE and while keeping the other participating components unaffected.

In this paper, we present the BROKER/SERVICES MODEL (B/SM), a system architecture model for CPE. B/SM provides abstractions to (formally) model the structure and behavior of CPE which can be subsequently made executable in a distributed environment. The basic constructs of B/SM are *brokers* representing the actors in a CPE (humans or software systems) and *services* defining the functionality of these actors. Brokers are *reactive* in the sense that they respond to *events* occurring in the CPE or its environment. The event-based interaction model yields a high degree of flexibility and ensures component autonomy, making it possible to evolve a CPE architecture with minimal impact on unmodified existing CPE-components. In addition, the reactivity of brokers is used to define and enact CPE-processes.

B/SM has initially been proposed for the software architecture of DBMS [11]. It subsequently was adapted to model CPE, assuming an active DBMS (ADBMS) as a process execution platform [21]. Since then, it has been extended to address heterogeneity of component systems and distribution within CPE. The contribution of this paper is to present a model and a corresponding runtime system that allows the integration of heterogeneous applications into a coherent and homogeneous cooperative process-oriented information system.

The remainder of this paper is organized as follows: in section 2, we describe the concepts of B/SM and exemplify their use in some detail. In section 3, we present the architecture of executable systems defined in the B/SM and implemented on top of a distributed event engine. In section 4, we briefly examine issues of component integration in a CPE. In section 5, we consider related work and finally, we conclude the paper in section 6 describing our intentions for future work.

2 The Broker/Services Model

This section introduces B/SM. We first list the requirements it is intended to meet and then present the model itself. A running example will be used to demonstrate various model elements (see Fig. 1). Consider a university department where students have to register for a diploma thesis. The registration is done by means of a forms-based application accessible to students. The students select a specific subject out of a list of available subjects which is kept on a department-specific object-oriented database. The registration can only be completed if the student has paid tuition fees for the current semester. Payments are recorded in a university-wide relational database. If the registration is successful, a confirmation of the registration is printed out at the student's terminal and at the department secretary's office which is then archived by a secretary. In our simple workflow, registrations may fail only if the tuition fees have not been paid.

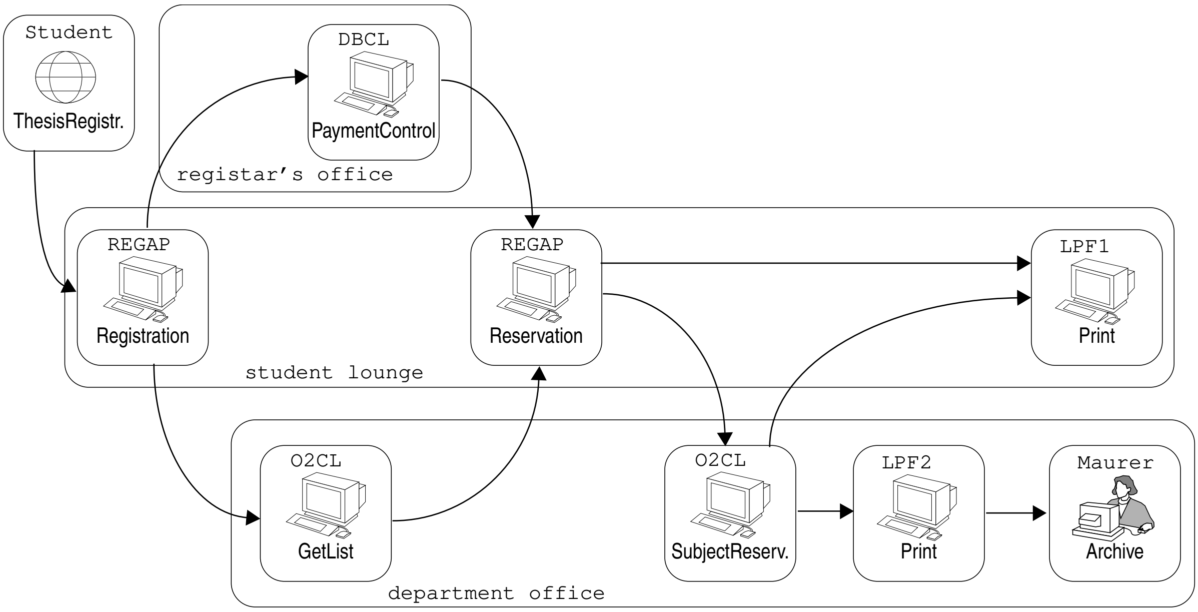

Figure 1. The example process activities and processing entities (including their location sites)

2.1 Requirements

The CPE-architecture model has to meet the following requirements:

- it must be capable to model component systems and the relationships between them
- it must support the representation of software systems as well as of people involved in processes (i.e., the notion of component should encompass software systems as well as representations of humans)
- it must model both, the static architecture as well as processes executable by the CPE (i.e., the behavior of component systems), and provide an integrated view of both aspects
- control and data integration must be supported
- it must support distribution of components with varying degrees of connectivity
- CPE-architectures should be easily made executable.

The last two requirements refer to the execution aspect of CPE which will be discussed in section 3. In this section we describe how the first group of requirements is met by B/SM.

2.2 Modeling the Software Architecture of CPEs with Brokers and Services

In this section, we introduce the elements of B/SM in more detail. For most of the elements, we present their syntax in abstract form and illustrate them through the running example.

2.2.1 Services

Services are functional abstractions and model the functionality of a CPE or its parts. A service has a signature, defining its name, a list of formal typed input parameters and a set of replies and exceptions. Default values may be given to request parameters during service specification which can be overwritten when the request is made. Service implementations are server-specific. Components thus interact by generating events and reacting to event occurrences in the system. In order to request a service, a corresponding *request event* must be generated. The service execution terminates with a *reply event* with a set of parameters including the execution result and data elements produced during the service execution. Abnormal service termination leads to the generation of *exception events*. Summarizing, a service is defined through its signature, a set of possible reply events (with parameters), and a set of possible exception events.

```
service service_name = {
    input_parameter*
    (reply reply_parameters)*
    exception_event* }
```

An example for a service would be the check if a registering student has paid the tuition fees:

```
service PaymentControl = {
    input    string studentNr;
    input    string semCode;
    reply    Paid(Date paydate);
    reply    NotPaid;
    raises   INVALIDPARAMETER;
    raises   SERVICENOTAVAILABLE; }
```

Note that the event-based communication mechanism leads to full server transparency: components can interact with each other and request services without having to know the eligible servers.

2.2.2 Brokers

A broker[1] models an actor in a CPE; it represents an instance of a software system, a component of the CPE infrastructure, or a person (e.g., a secretary involved in a workflow). Generally, a broker is characterized by its state, its behavior, and its interaction with the environment. These elements are now successively introduced, and the syntax of broker definitions will be given in section 2.2.3. The characteristics of brokers depend on the properties of the real-world entities they represent. This gives rise to a broker typology described in the following section.

The broker state is a collection of interface declarations. The type system usable for defining a broker state is based on the OMG IDL [17]. State information typically includes the interfaces of external data elements used by a broker to execute its tasks. For example, in order to access an object-oriented database, the broker state will include a database entry object and the messages this object can understand.

As a further example, consider a broker `LPF` managing the queues of two printers (`lq1`, `lq2`). Its state is defined as follows:

```
interface PrinterQueue {
    enum     OpResult {ok, error};
    string   name;
    unsigned append_job(string file);
    unsigned cancel_job(unsigned job_id);
    unsigned queue_state(); };

broker LPF2 = {
    state
      PrinterQueue lq1;
      PrinterQueue lq2;
    ... };
```

1. The notion of “broker” should not be confused with “object request brokers” as proposed by the Object Management Group (OMG). For a discussion of this issue see section 5.

Brokers are characterized by their interaction with their environment, i.e., with other brokers. Interaction in B/SM is performed by generating and detecting events. Thus, brokers are *reactive* in the sense that they can react to events occurring during the execution of a process in the CPE. Brokers interact with their peers whenever they need a service available in the system (which is provided by one or more other brokers). This interaction is modeled by an event representing the service request and another event (reply or exception) representing the result of the service execution. Additionally brokers may raise exceptions which are defined at a system level and not for a specific service.

The reaction of a broker to events (the *behavior* of the broker) is defined by one or more event-condition-action rules (ECA-rules), as proposed for active database management systems [1, 22]. The set of ECA-rules of a broker is also referred to as the broker's *capabilities*. Intuitively, the meaning of rules is that whenever a broker detects an event, it checks whether the condition holds and—if it holds—executes the action. ECA-rules are used to define how brokers implement services, i.e., they define the reaction of the broker to a service request. In this case, the event part of the ECA-rule corresponds to a request event (of the particular service). Likewise, in case a broker requires the result of a service execution for further processing, it will define one or more rules reacting to the reply event. The event part of this rule then corresponds to the reply event in question. The syntax of ECA-rules is as follows:

```
capabilities { rule* }
rule ::=  on [name =] event
          [if condition]
          do actions
```

The situations to which a broker has to react can be much more complex than simple requests or replies. For instance, the situation can be defined as the occurrence of multiple reply events. Such situations are modeled in B/SM by *composite events* similar to those proposed for active database management systems [e.g., 7, 9]. A composite event is constructed out of primitive or other composite ones by using one of the following constructors:

- *sequence*: the component events have occurred in the specified order,
- *disjunction*: one of the component events has occurred,
- *conjunction*: the components have occurred in any order,
- *repetition*: the component event has occurred a predefined number of times,
- *negation*: the component event has not occurred within a predefined time interval,
- *concurrency*: the component events have occurred simultaneously.

Several of the constructors require the specification of a monitoring interval, which in turn is given as a pair of events (e_1, e_2). An event `e` is said to occur within the monitoring interval if `e` occurs after e_1, and e_2 does either not occur at all or after `e`.

Additionally, *composite event restrictions* can be formulated. For several constructors (sequence, conjunction), multiple occurrences of a certain component event type might be eligible upon event composition. In such cases, it can be defined that the component events should have been generated by the same component and/or within the same workflow. Such restrictions are expressed in ECA-rule definitions by adding the phrase `same origin` and/or `same workflow`.

The proper semantics of composite events depends on the ordering of event occurrences. Ordering is temporal, i.e., is based on the timestamps of the event occurrences (which in turn is the point in time when an event is detected). The mechanism to establish such a temporal ordering in a distributed environment [20] extends the work described in [19].

An ECA-rule condition is a boolean expression. It can check the values of state variables of a broker, or request and reply parameters, and it can invoke operations defined for the broker which return a boolean result. The action part is defined using a programming language (C++), with several additional pre-defined operations (`request`, `reply`, `raise, enqueue, dequeue).` The `request`-operation is used to express a request for a service. `Reply` generates a reply event and represents the result of a service execution, and `raise` generates an exception event, meaning that the service cannot be provided out of some reason.

As an example, consider the ECA-rule below, implementing the `checkSubject`-service. According to its event part, the rule fires whenever the service `CheckSubject` is requested. The first four lines in the action part execute a transaction in which a persistent object is read from the database, and a message is invoked for this object. The last two lines show the usage of the `reply`-operator. Note that the action part is in our case implemented using the language binding of the underlying object manager (i.e., Shore [5] from the University of Wisconsin).

```
on   request(CheckSubject, string subjectId);
do   {
        SH_BEGIN_TRANSACTION(shorerc);
        shorerc = Ref<Subjects>::lookup(subjectMgr, tmgr);
        res = tmgr->stillAvailable(subjectId);
        SH_DO(SH_COMMIT_TRANSACTION);
        if (res)
           reply(SubjectFree);
        else
           reply(SubjectNotAvailable);
     };
```

2.2.3 Broker Typology

A typical CPE and its processes will comprise both, automated as well as manual tasks. It is thus a requirement to model not only software systems but also to represent humans. Moreover, organizational relationships can exist between people, which may also behave differently in different contexts. As a consequence, the model should be able to distinguish between the various notions of "component", and should be able to handle the different kinds of information that are available for different types of components.

These requirements are met by B/SM by distinguishing between different specializations of brokers (see Fig. 2 for the broker specialization hierarchy in UML-notation). In general, brokers are distinguished in *external brokers*—representing entities that belong to the CPE-ex-

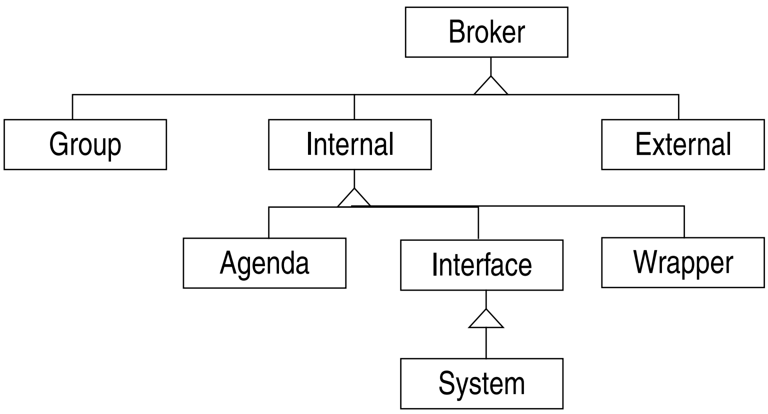

Figure 2. The broker typology hierarchy (in OMT notation)

ternal world, *internal brokers* which are part of the CPE, and *group brokers* consisting of groups of brokers. External brokers provide the stimuli which arrive from outside the system (e.g., the initiation of a process) by declaring a set of events that they generate. External brokers do not provide any service to other brokers. Services in a CPE are provided by the different kinds of internal brokers. Group brokers compose organizational entities consisting of other broker types.

Internal brokers represent processing entities—people or software systems—which participate in the workflows executed in the workflow system. System brokers homogenize heterogeneous collections of processing entities by mediating between the homogeneous B/SM world and the functions of the represented system. They are specialized into interface brokers, agenda brokers, and wrapper brokers.

An *interface broker* acts as a representative of an open software system which is accessible to the "outside" world through a programmatic interface or an extension language. In other words, an interface broker actually renders the functionality of this system accessible to other brokers by making use of the interface of the represented system. For interface brokers, it is known which services they provide and how these are executed in terms of ECA-rules as well as the services they can request. In the action parts of system broker ECA-rules the access to the interface of the represented system takes place (e.g., calls to the system's application programming interface). An example of an interface broker is a wrapper for an Oracle-application providing an interface function for checking the tuition payments of students. A specialization of interface brokers are *system brokers.* The services provided by a system broker are fully implemented within the CPE, and use internal data. For system brokers, it is known which services they provide and request. It is also known how the services are provided in terms of ECA-rules and implemented operations. An example of a component broker is a clock signaling the system time.

Wrapper brokers specialize internal brokers by representing software systems which perform activities in the workflow system, do however not provide a programmatic interface. Access to these systems is possible through protocols described in *envelopes*.

Agenda brokers are another specialization of internal brokers. They represent the access interface for people which perform manual tasks in a CPE. Information about the services a person provides and the ones he may request is available. In contrast to interface brokers, knowledge about the service implementation is not available, i.e., behavior cannot be entirely specified—otherwise, the task could be automated. Information on how people accomplish their tasks is expressed by simple ECA-rules whose action parts only append (operator `enqueue`) to or remove (operator `dequeue`) requests from the user's worklist. Agenda brokers are useful for the integration of manual tasks and responsible entities into a system.

Group brokers are a further specialization of brokers. Their semantics is to represent organizational structures consisting of subgroups and people (e.g., project teams). They are eligible participants in organizational relationships. Group members can belong to more than one group at a time. No behavior can be defined for groups. Groups however, may have a state accessible to their members. Considering our example, we have the following brokers:

- the interface broker `O2CL` which is a client of the object-oriented DBMS O_2 in which the list of available subjects is stored
- the interface broker `DBCL` which represents the client of a relational DBMS in which administrative information about students (e.g., tuition payment information) is managed
- the wrapper brokers `LPF1` and `LPF2` wrapping printer spoolers
- agenda brokers represent each involved secretary (`Maurer` and `Ebner`)

Table 1. Kinds of brokers and knowledge available about them

Broker Type	*State*	*Services Provided*	*Services Requested*	*Behavior (ECA-rules)*	*Implementation*
external	no	no	yes	no	no
group	(yes)	no	no	no	no
agenda	yes	yes	yes	(yes)	no
wrapper	yes	yes	yes	(yes)	envelopes
interface	yes	yes	yes	yes	interface calls
system	yes	yes	yes	yes	yes

- a group broker (`Secretaries`) models all secretaries working in the department
- an external broker (`Student`) which triggers the registration process.

2.2.4 Contexts

In general, a processing entity can play different roles in a workflow. It will react differently to requests depending on the context in which the request was generated, or depending on the role the component is assumed to play when serving the request. B/SM provides a construct for the packaging of workflow-specific behavior called a *context*. A context consists of a set of one or more ECA-rules and state variables:

```
context identifier = {
      state_decl* interaction_decl* rule_decl*
};
```

Contexts have been proven useful for the description of complex process situations described in high-level workflow specification languages. Such languages are more adapted to the description of processes abstracting from concrete execution issues. In our ongoing work we use the workflow specification language TRAMs [14]. The transformation of a TRAMs-specification to an executable system representation in B/SM takes place by the following general steps:

1. The workflow designer retrieves from the Broker/Services repository the specifications of processing entities (brokers) and the services they provide.
2. Services correspond to activity types in TRAMs workflows whose input parameters correspond to the request event type parameters in the B/SM specification. The possible replies in B/SM are mapped to end sub-states in TRAMs activity specifications.
3. The workflow designer then uses the constructs provided in TRAMs to define the control flow between these activities, define workflows and instantiate activities and workflows.
4. The TRAMs specification is then transformed to an executable representation of the system architecture which contains complete broker definitions with capability rules pertaining to the workflow type transformed, contexts containing the workflow type control flow expressed in ECA-rules, responsibilities relating services to their providers, etc.

2.2.5 Organizational Relationships

In a typical organization, the involved people will be associated by various organizational relationships. Common examples of such relationships include supervisor/supervisee relationships, group memberships, etc. Such relationships are often important for determining the assignment of tasks, namely whenever the entity that must execute a service depends on who executed previous activities. For instance, it can be required that if service A has been executed by a clerk, then service B must be executed by the manager of the clerk. Furthermore, different organizations usually have varying organizational structures and relationships. Thus, it is not sufficient to only provide a fixed set of relationships with predefined semantics.

In B/SM, general relationships between brokers can be used to represent organizational relationships. They are general in the sense that no specific semantics is associated with them.

The syntax of organizational relationship definition is:

```
relation identifier = { from_broker rel_name to_broker };
```

As an example, both secretaries belong to the group `Secs`:

```
relation rel1 = { Maurer member AdminStaff };
relation rel2 = { Ebner member AdminStaff };
```

2.2.6 Responsibilities

As mentioned above, services are requested and provided by brokers. Often brokers do not provide services or—more generally—react to events unconditionally. Instead, depending on the concrete workflow context in which an event occurs, different brokers might react. This kind of information is specified by *responsibilities*. A responsibility relates an event type and a set of brokers. A formula can be specified that restricts the responsibility of the brokers. The atomic predicates usable in such a formula can refer to event parameters, the broker state, and organizational relationships. The syntax of responsibility definitions is the following:

```
responsibility identifier = { event_name, brokers, context [, formula] };
```

As an example, consider the secretary being responsible for the registration service:

```
responsibility registrationResp = {
    doRegistration, Maurer, thesisRegistrationContext };
```

2.2.7 Support for Processes

One of the requirements mentioned that the model should also support processes. One alternative would be to support the explicit notion of "workflow" and to interpret workflow specifications. However, this approach tends to be rather inefficient. In the B/SM framework, workflows exploit the reactiveness of brokers. Thus, workflows are implicitly specified by defining the situations in which particular brokers have to react. This is straightforward in the case of activity execution; these are modeled as a service request/reply pair. B/SM also is able to express more complicated cases of control flow such as and-joins which can be modeled as composite events. As an example, consider the condition that if the activities `checkSubject` and `checkFee` have been successfully executed, then the activity `doRegistration` should be performed. This is expressed in B/SM as the following ECA-rule for the `thesisRegistrationContxt`:

```
on   reply(FeePaid) && reply(SubjectReserved): same workflow
do   request(DoRegistration);
```

Note that these additional ECA-rules specifying the behavior of a broker within a workflow define *workflow-type-specific* control flow, while those defining the reaction to requests or replies define the *workflow-independent* broker behavior (see also 2.2.4).

Finally, it has to be defined how workflows are started, and what are the possible terminations. For that matter, B/SM treats workflows as a special kind of services, in that the initiation of a workflow by a start event generates a series of request events and the termination of a workflow corresponds to a reply event. Workflows that cannot be executed successfully can generate exceptions.

```
workflow type identifier = {
     start request
     requests request*
     end reply* exception*
}

workflow type thesisRegistration = {
     start
        Register(String subject, String student, Semester semester);
     requests
        CheckSubject(String subject);
        PaymentControl(String student, Semester semester);
     end
        Success; Failure;
};
```

2.3 Well-defined CPE-Architectures

A CPE-architecture is defined by a collection of broker definitions (including contexts and process-specific rules), service definitions, and responsibilities. The B/SM-model allows to express several constraints on “nice” CPE-architectures. We require that an architecture is *complete* in the sense that functionality needed by some components is actually provided by one or more other ones. *Minimality* refers to the fact that only those functionality is incorporated in the CPE that is actually required by a component. Finally, *syntactic correctness* refers to the fact that the broker implementations of services using ECA-rules obey the interfaces specified by the service definition.

These constraints can be expressed in B/SM as follows. Completeness requires that

- for each service which is requested by some component, there is at least one corresponding responsibility defined for at least one broker,
- for each responsibility defined for a broker, it contains at least one ECA-rule implementing the service,
- for every reply (except termination events), there exists at least one matching ECA-rule catching the event defined in a broker or a context.

Minimality holds if each service for which a responsible broker has been specified is requested by at least one broker. A violation of the minimality constraint can have two causes. The first one is that superfluous functionality has been specified. The second one is that semantically, the offered service is indeed needed by another component, but corresponding requests have been erroneously omitted.

Finally, syntactic correctness holds (statically), if the requests raised by brokers comply to the service definition, and if reply events raised by service providers comply to the signature of the reply as defined in the service definition.

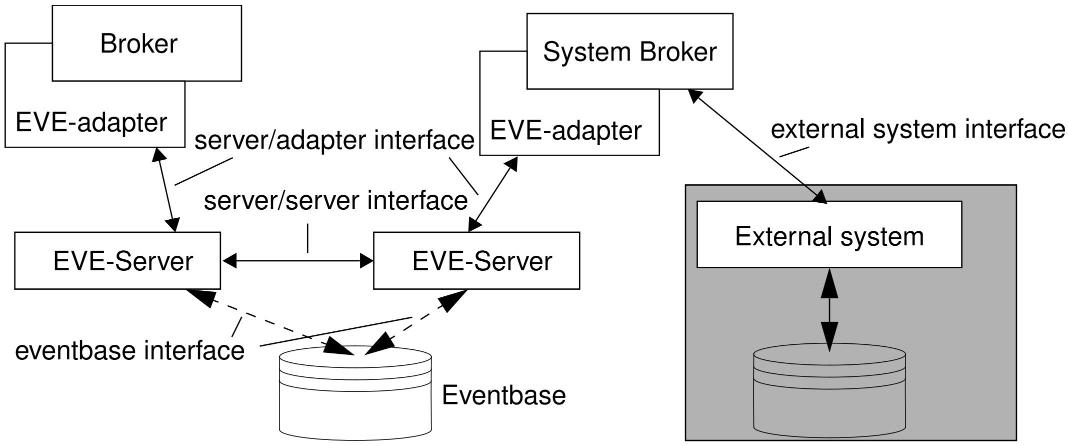

Figure 3. Conceptual architecture of EVE and brokers

3 Executable CPE-Architectures

3.1 Broker Execution Platform

CPE-architectures specified in B/SM can be made executable through the so-called **Ev**ent **E**ngine (EVE). EVE is described elsewhere [12], so that we discuss it here only briefly.

EVE is a distributed, multi-server system (see Fig. 3) targeted to the execution of workflows. It implements event detection, event history maintenance, event notification to brokers, and task assignment for workflows. EVE is implemented on top of Shore [5], i.e., it uses Shore in order to persistently store information. EVE's internal operations are implemented as Shore-transactions in order to ensure recoverability and serializability of concurrent client requests.

EVE has a multi-server distributed architecture. There is one EVE-server per local network, and multiple servers are assumed to be connected to a wide area network. EVE supports ECA-rules, where the supported event types conform to those listed in section 2.2.2, and condition and action parts are formulated in Shore's C++ language binding. EVE-servers are able to compose events out of components having occurred at different sites.

3.2 Broker Execution

A broker is represented in EVE by an *EVE-adapter*. EVE-servers and brokers communicate via their EVE-adapters which maintain event queues for their brokers. Thus, in case a broker is temporarily disconnected from the system, its events will be kept in this queue.

Brokers are made executable in different ways, depending on their type. System and automatic brokers are compiled into dynamically linkable libraries. Automatic brokers execute their functions under control of their local EVE-server. The functions of system brokers are executed by a demon, which must be running on every machine which is part of the CPE. Each agenda broker is represented by a user interface, whose most important state component is a work list. These interfaces also provide the functionality to exchange events between the broker and the execution platform. A framework for the construction and customization of these user interfaces in Java is currently under development.

Brokers execute their operations according to the following protocol:

1. Each event of interest for a broker is added to the event queue of the corresponding broker.
2. The broker is informed about the event occurrence:

i. For an agenda broker, it is checked whether the broker is currently connected to its EVE-server (i.e., the person is logged in the system). If yes, the broker is notified that a new event has arrived for it.
ii. For a system broker, the corresponding demon is notified about the event occurrence.
iii. An automatic broker will directly react to the update of its event queue (i.e., no extra notification is necessary).

3.3 Operational Semantics

In the course of execution of a CPE, brokers generate events. Other brokers react to them, thereby generating new events. These occurrences are ordered according to the time of their detection and are stored in the *event history*. Thus, the behavior of a broker can be characterized by those event history elements it generated. Likewise, a workflow is characterized by a subsequence of the event history. In other words, event histories as generated by brokers can be used to define the semantics of brokers and CPEs (see [20]).

Table 2 shows an event history generated during an execution of the example workflow.

Table 2. Events generated during sample workflow

Event	Type	Parameters	Origin
Register	request	subjectId, studentNr, semCode	external
CheckSubject	request	subjectId	Maurer (agenda)
PaymentControl	request	studentNr, semCode	Maurer (agenda)
FeePaid	reply	paymentDate	DBCL (interface)
SubjectFree	reply		O2CL (interface)
SubjectFree && FeePaid	conjunction		EVE (system)
DoRegistration	request	subjectId, studentNr	Maurer (agenda)
Registered	reply		Maurer (agenda)
PrintFile	request	registrationFile	Maurer (agenda)
Success	reply	printerName	LPF (wrapper)

4 The CPE Component Integration Process

In this section, we briefly describe the construction process of CPE. In particular, we discuss which information has to be provided by the CPE-designer, which information is generated, and which information can be reused for different CPE-constructions. The construction process consists of three phases:

1. the specification of workflow types,
2. the derivation of a B/SM-specification,
3. the transformation into a form executable by EVE.

Specifications of workflow types define the structure of workflows (i.e., which sub-workflows and atomic activities they contain), processing entities (including roles and organizational entities), assignments of processing entities to activities, control flow (i.e., in which order the activities and sub-workflows have to be executed), and temporal constraints such as deadlines.

In the second phase, a specification in B/SM is derived from the workflow specification. The following mapping rules are used:

- atomic activities are mapped onto services,
- processing entities are mapped onto brokers,
- task assignment rules are mapped onto responsibilities, and
- control flow is mapped into ECA-rules with possibly composite events.

For each broker, the following parts have to be defined:

- the state,
- contexts,
- the ECA-rules implementing the services for which the broker is responsible, and
- ECA-rules implementing the process-specific behavior.

These parts have to be implemented manually, unless previous specifications can be reused. In fact, typical workflow specifications will contain artifacts that have already been used before in other specifications. The B/SM model lends itself well to the reuse of broker and service specifications and implementations, and thus frees the designer from implementing brokers repeatedly from scratch.

Reuse of broker implementations is accomplished by storing workflow specifications and broker implementations in a repository. This repository also associates processing entities with corresponding brokers. Therefore, during the derivation of a B/SM specification, those brokers can be reused that are needed to implement one of the processing entities and are contained in the repository.

In the third step, the B/SM-specification is compiled into EVE-data structures and rules.

5 Related Work

B/SM is related to work in different areas; for the sake of conciseness we discuss only the most important ones. We compare the related approaches with the ability of B/SM to model and operate cooperative, process-oriented information systems.

Not just because of the similar terminology B/SM is related to CORBA [17]. The concept of broker in B/SM corresponds to application objects in CORBA, and an object request broker in CORBA would correspond to EVE in our approach. The major difference is the reactiveness of B/SM-brokers, i.e., the ability to react to possibly complexly defined situations. The prime interaction mechanism in CORBA is method invocation, while the event-based architectural style of B/SM leads to a looser integration and eases integration and evolution of components. The CORBA-architecture defines a notification service for primitive events only, and thus the notion of a reactive component able to react to complexly defined situations is not directly supported.

Using “events” as a means for software integration has been proposed in several areas (e.g., the notification service of CORBA, software engineering [4], and the Tuxedo transaction processing monitor [2], to name only three). However, these approaches usually have in common that they support primitive events only, while composite events play an important role in B/SM especially for workflow management.

The novelty of B/SM with respect to workflow management is that it allows to integrate processing entities into a coherent architectural view. While workflow management systems such as Flowmark [16] are usually able to execute application programs under certain restrictions, the B/SM approach based on wrapping is less restrictive and in addition provides designers with a rigid notion of software architecture (which is mostly missing in current workflow management systems).

Several approaches towards workflow management also propose the use of events (pioneered in [8] and subsequently used by others [e.g., 6, 15]). B/SM and EVE present, however, the first approach integrating event-driven workflow execution with a notion of system architecture. Additionally, those other approaches use only primitive events and thus have to resort to "passive" approaches for more complex situations, like and-join control flow dependencies. In INCAS [3], composite events can be used to define the behavior and routing of so-called information carriers, while the semantics of these rules in a distributed environment remains unclear.

6 Conclusion

In this paper, we have introduced the BROKER/SERVICES MODEL, and have sketched how it can be used as an integration framework for heterogeneous and process-oriented environments. This model is particularly well-suited to model the actors in such environments, e.g., database applications or humans. It also allows the definition of workflows among the CPE-components. The usage of events as the major mechanism for interaction and workflow management leads to a maximal degree of flexibility of CPE-architectures and autonomy of CPE-components. Finally, CPE-architectures can be transformed into executable systems, using EVE as a runtime system.

Notwithstanding the expressive power of B/SM, there are also restrictions that will be addressed in our future work. For instance, B/SM supports flexible mechanisms for task assignment (based on responsibilities and contexts). However, once a request has been assigned to a particular broker at runtime, this assignment remains fixed and it is not possible to "re-route" the request. This will be addressed by allowing brokers to delegate requests to other brokers, and by facilities to define in which situations delegation is legitimate.

Another important issue that we will address in the future is the specification of service and workflow execution cost and time. Cost and execution time information can then be used to schedule workflows more flexibly and, e.g., to apply workflow escalation [18] in order to complete workflows within a given time-frame.

7 References

1. The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM SIGMOD Record* 25:3, September 1996.
2. BEA TUXEDO. The Programming Model.. White Paper, BEA Systems, Inc., November 1996 (http://www.beasys.com/products/tuxedo/tuxwp_pm/tuxwp_pm1.htm).
3. D. Barbara, S. Mehrotra, M. Rusinkiewicz. INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical Report, Matsushita Information Technology Laboratory, Princeton, May 1994.
4. D.J. Barrett, L.A. Clarke, P.L. Tarr, A.E. Wise. A Framework for Event-Based Software Integration. *ACM Trans. on Software Engineering and Methodology* 5:4, October 1996.
5. M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T.

Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, M.J. Zwilling. Shoring Up Persistent Applications. *Proc. 1994 ACM SIGMOD Intl. Conf. on Management of Data*, Minneapolis, May 1994.

6. F. Casati, S. Ceri, B. Pernici, G. Pozzi. Deriving Active Rules for Workflow Management. *Proc. 7th Intl. Conf. on Database and Expert Systems Applications*, Zurich, Switzerland, September 1996.
7. S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. *Proc. 20th Intl. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.
8. U. Dayal, M. Hsu, R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Atlantic City, NJ, May 1990.
9. S. Gatziu, K.R. Dittrich. Events in an Active Object-Oriented Database System. *Proc. 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, UK, August/September 1993.
10. D. Georgakopoulos, M. Hornick, A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, Kluwer Academic Publishers, September 1994.
11. A. Geppert. Methodical Construction of Database Management Systems. Doctoral Dissertation, University of Zuerich, Switzerland, 1994.
12. A. Geppert, D. Tombros, K.R. Dittrich. EVE, an Event Engine for Workflow Enactment in Distributed Environments. Technical Report 96.05, Department of Computer Science, University of Zurich, May 1996.
13. S. Jablonski, C. Bussler. *Workflow Management. Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London 1996.
14. M. Kradolfer, Andreas Geppert. Modeling Concepts for Workflow Specification Technical Report 97.05, Department of Computer Science, University of Zurich, May 1997.
15. G. Kappel, P. Lang, S. Rausch-Schott, W. Retschitzegger. Workflow Management Based on Objects, Rules, and Roles. *Bulletin of the IEEE Technical Committee on Data Engineering* 18:1, March 1995.
16. F. Leymann, W. Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal* 33:2, 1994.
17. The Common Object Request Broker: Architecture and Specification. Revision 2.0. The Object Management Group, July 1995.
18. E. Panagos, M. Rabinovich. Predictive Workflow Management. The Third International Workshop on Next Generation Information Technologies and Systems (NGITS '97), Neve Ilan, Israel, June/Luly 1997.
19. S. Schwiderski. *Monitoring the Behavior of Distributed Systems*. PhD Thesis, University of Cambridge, 1996.
20. D. Tombros, A. Geppert, K.R. Dittrich. Semantics of Reactive Components in Event-Driven Workflow Execution. *Proc. 9th Int'l Conference on Advanced Information System Engineering*, Barcelona, Spain, June 1997.
21. D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C.B. Jones, C. Lengauer and H.-J. Schek (eds.), *Object-Orientation with Parallelism and Persistence*, Kluwer Academic Publishers, 1996.
22. J. Widom, S. Ceri (eds). *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1995.