Brokers and Services: Constructs for the Design and Implementation of Process-Oriented Environments

Dimitrios Tombros Andreas Geppert Klaus R. Dittrich

Technical Report 95.26

Institut für Informatik, Universität Zürich
Winterthurerstr. 190, CH-8057 Zürich, Switzerland
Email: {tombros | geppert | dittrich}@ifi.unizh.ch

Abstract

Cooperative process-oriented environments (CPEs) are federations of human agents and multiple (heterogeneous) software tools which cooperate to accomplish complex and long-lasting tasks. Appropriate abstractions must be available in order to properly design and implement CPEs, including tasks such as the control of and cooperation among their participating entities. In this paper, we introduce the concept of *brokers*, which are components reacting to environmental stimuli and are used to model participating entities in CPEs: human participants, existing software tools, or parts of the environment infrastructure. *Services* model the functionality offered by brokers. Although these brokers are actually abstractions on top of active object-oriented database management systems, they are also influenced by concepts developed in other research areas, such as (concurrent) programming languages, artificial intelligence, and software architectures. We therefore also position the broker model with respect to similar concepts from these domains.

Keywords: active database management systems, cooperative computing, software processes, workflows, system construction

1 Introduction

Cooperative process-oriented environments (CPEs) must support the execution of long-lasting processes involving several human agents and software tools. The behavior and interaction of the components of these environments is described through an explicitly defined *process model (PM)*. The processes modeled consist of steps dependent in various ways among each other. Examples of CPEs are workflow management systems [16] and process-centered software development environments [10].

The construction of CPEs gives rise to challenging requirements due to their dynamic and highly complex nature. In particular:

- a CPE must be able to control and execute long-lasting processes,
- the architecture of a CPE should be flexible, in order to seamlessly integrate different components and allow the execution of application dependent processes,
- heterogeneous components, often developed without the intention to cooperate, need to be integrated in a CPE,
- in order to keep process execution times as short as possible, process steps should be executable concurrently (unless serialization of steps is required due to dependencies),
- designs and implementations of *CPE-artifacts* should be reusable (PM elements, component designs and implementations, etc.).

In this paper, we describe a metamodel for the design and the implementation of CPEs: the *broker/services model* which is intended to provide an appropriate construction formalism for CPEs fulfilling the aforementioned requirements. Brokers describe reactive components responsible for providing specific services. Their interaction occurs based on implicit invocation mechanisms [13]. Brokers realize the execution of processes by reacting to specific predefined situations such as process states or deadlines. They are autonomous and self-contained, allowing the flexible and modifiable definition of a CPE-architecture and promoting their reusability in different CPEs. Autonomy is achieved by defining broker reactions to situations of interest within the brokers themselves.

In order to react appropriately, brokers have to monitor any process execution they are involved in. Furthermore, since processes may have a long duration, their state and the data they use must be made persistent (and recoverable). We thus implement brokers on top of database management systems (DBMSs). Concretely, we make use of an active, object-oriented DBMS (ADBMS) [9], which is then also used for monitoring the process state and for notifying brokers when situations arise where they have to react.

This paper is organized as follows. The next section discusses requirements for CPE-architectures in more detail. Section 3 introduces the broker/services model, section 4 shows how these concepts are applied to model CPEs, and section 5 describes the current implementation on top of a concrete ADBMS. Section 6 compares our approaches with others, including those from related research areas, and section 7 concludes the paper.

2 Requirements for a CPE Construction Formalism

In this section, we identify the requirements for a CPE construction formalism.

2.1 Process Support

A CPE has to support the notion of "process" (in the sense of software processes/design processes or general workflows). A *process* is a long-running activity consisting of multiple *steps*; a step is either atomic or a (sub)process. The execution order of steps can be constrained by conditions over required execution states, related input/output data, or temporal relationships between steps. The structure and behavior of a process is defined in the corresponding PM. Process support refers to two subtasks:

- process modeling, and
- process model enactment.

The first task refers to defining the PM in question, including its structure in terms of steps and components involved, as well as modeling the operational aspect in terms of dependencies between its subprocesses or steps. *PM evolution* should be supported, meaning that changes in the real-world can be accommodated by modifying the PM accordingly.

Second, PMs must be implementable. The execution of processes requires that the CPE is able to monitor the state of running processes and to control process execution. In case of process evolution, it must be ensured that executing processes are consistent with the modified PMs.

2.2 Flexible Software Architecture and Customizable CPEs

Different PMs — including different sets of components involved in processes — lead to *different CPE-architectures*. Support is thus needed for the systematic customization of the CPE-behavior as required by the PM, and for the integration of the components referenced in the PM into the CPE-architecture. Such a CPE-framework (or *CPE-metamodel*) should be object-oriented, since the kind of persistent data produced e.g. during software processes can be best stored in object-oriented database systems [7]. However, in addition to "objects" a notion of *component* is required for enabling the flexible construction of open CPEs (for the need of components in the construction of general open systems, see [25]). In CPEs, components are required for the following reasons:

- objects delegate tasks by sending messages to other objects, while more flexibility can be achieved when clients do not have to know receivers of messages,
- objects react only when they receive messages, while more powerful reaction mechanisms are needed (e.g., automatic reaction when specific process states are reached).
- the granularity of objects in their conventional use in object-oriented programming is too fine for modeling coherent CPE-parts.

A CPE-framework thus has to support the concept of a component which represents a coherent (sub)part of a CPE able to react to process states. The CPE-framework should also satisfy changing application domain requirements and CPE-evolution as well as component reusability. More specifically these requirements can be expressed as follows:

- additional components should be easily added or removed from the system, and
- the functionality of existing components should be modifiable without extensive effort.

Thus the modeling of components should be done in a way emphasizing *what* functionality a component provides instead of *how* this functionality is realized. Also, the modeling of component interaction should be kept as simple as possible and should allow the integration of additional components without affecting already existing ones. To achieve this, component interaction should not be encoded in the components themselves [13].

The customizability requirement can be extended to encompass the integration of external tools. Tools developed without concern for their use in a specific CPE often have to be seamlessly integrated either due to compatibility issues or because their functionality is expensive to reimplement. Thus, the CPE-framework must provide integration mechanisms to encapsulate external components and provide their full functionality to other system components. These mechanisms should also hide tool idiosyncracies such as proprietary interfaces or tool-specific transaction models.

2.3 Transactions: Concurrency Control and Recoverability

A CPE should support transaction management with various functionalities. First, steps of a process should be executable as atomic and isolated transactions. Thus concurrency control and recovery should be provided for the steps of a process. Unless prohibited by the dependencies defined for a process, several steps should be concurrently executable. Second, components should be able to serve several requests simultaneously. In this way, components are internally concurrent and can execute multiple steps, which can belong to the same or different processes. Finally, some transaction properties should be definable/specifiable for (entire) processes. In this way, processes are long-lasting transactions (e.g., in the sense of Sagas [12]), they can be isolated from other concurrent processes, and executing processes can be aborted and recovered.

3 The Broker/Services Model

In this section, we introduce the broker/services model (the CPE-metamodel) we use to describe the structure and behavior of a CPE. In order to clarify the concepts presented, we will use throughout the next sections parts of an example of CPE-modeling with brokers from the domain of software development environments. The complete example is presented in the appendix. Our example represents a small fragment from an edit-compile-test cycle. Another example from the workflow management domain can be found in [18].

3.1 Overview

A CPE consists of several participating interacting and autonomous component entities called *brokers*. Brokers are reactive components offering *services* to other brokers (their clients). Their behavior is defined in terms of (event-condition-action) rules, they react automatically to simple events such as requests made by other brokers or to complex process states. Brokers can behave in various ways in different contexts. They thus can have roles defined describing the context-specific behavior. A PM is mapped to a collection of brokers and event-condition-action-rules (ECA-rules) defined for the brokers. In our model real-world processes are implemented by sequences of actions performed by brokers. PM enactment is initiated when the first action in a PM is executed by a broker. Finally, brokers are also responsible to manage the production or design data they need to perform their tasks, and thus can possess an internal state. In Fig. 1 an overview of the concepts of the broker/services model is presented. Fig. 2 visualizes the CPE-metamodel architecture.

```
BROKER broker_name: broker_kind
   [STATE {state_name: type}]
RULES
   {RULE rule_name
   ON event
   [IF condition]
   DO action
   [PRECEDENCE rule_name_list]}
{ROLE role_name
   {//rules as above }}
```

```
SERVICE service_name
  PARAMETERS {param_name: type}
  REPLIES
  {reply_name, {param_name: type}}
  EXCEPTIONS {exception_name}
```

```
RESPONSIBILITY service_name: broker_name
```

Figure 1: Key Concepts of the Broker/Services Model (keywords are written in upper case, brackets enclose optional terms, and curly brackets enclose terms that may appear zero or more times)

The principal brokers involved in the example include a `SoftwareEngineer` playing the role of a programmer, an `Editor`, a `UserInterface`, a `Compiler`, and an `ObjectManager`. Their cooperative work starts when the `Programmer` receives a request to modify the code of an existing module. He then requests a new `Editor` session. The module file is checked-out into the `Editor's` private workspace and displayed into an `Editor` window provided by the `UserInterface`. Once the `Programmer` has completed the changes, he starts the `Compiler` through an `Editor` command. If the compilation is error-free, the module file is checked-in back to the `ObjectManager's` shared repository and can be unit-tested.

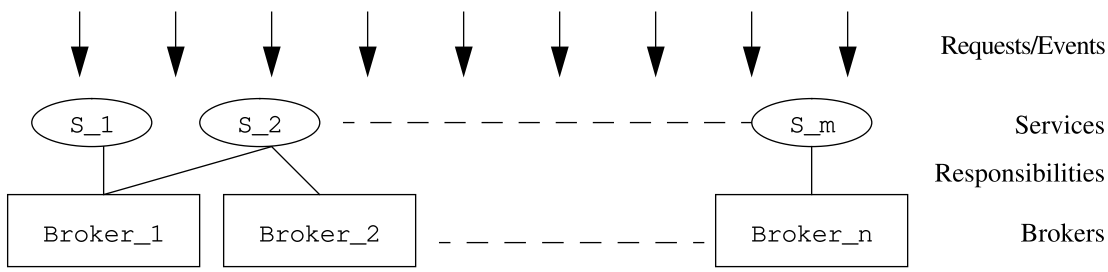

Figure 2: Relationships between Brokers, Services, and Requests/Events

3.2 Services and Responsibilities

The functionality of a given system is described by a set of *services*. A service is specified by a service signature consisting of the service name, its parameters, the possible replies and exceptions its request may cause. A specific service is provided by one or more brokers and can be requested by various client brokers. The association between a service and its providers is established by means of m:n relationships called *responsibilities*.

Services represent the external interface of brokers, and either can be of interest to end-users of the system (e.g., providing the capabilities of the user interface or application functionality), or can be used by other components to achieve their functionality (e.g., by bundling simpler services into more complex ones). By using services to describe the capabilities of an environment, the description of the work done can be abstracted from the components actually performing it and consequently from the system implementation.

The services needed in the example process include `ModifyCode` which initiates the whole work, `Edit` provided by the `Editor`, `Display` provided by the `UserInterface`, `CheckOut` of files provided by the `ObjectManager`, and `Compile` provided by the `Compiler`.

3.3 Brokers

Brokers are self-contained system components capable of autonomous reactive behavior, and responsible for the provision of specific services. In order to describe varying degrees of component integration in a given system (and thus knowledge about the internal structure and implementation of components), we distinguish between three kinds of brokers: *internal*, *external,* and *interface* brokers. This characterization of brokers depends on the following factors viewed from a system modeling perspective:

- knowledge of their implementation and structure, i.e. of the way they provide the services they are responsible for,
- knowledge of their functionality, i.e. of the services they provide to other system components (which may in turn be in part realized by services they request themselves),
- knowledge of their interaction capabilities with other components, i.e. of the services they may request from other components.

For internal brokers, we completely specify the interface and the internal processing logic. For interface brokers, we only specify their interface to other components by defining the services they provide and the services they request. Finally, for external brokers, we only specify the possible requests they might make from other system components and other situations of interest they might generate. The decision which kind of brokers to use when modeling a system thus depends on the modeller's knowledge about a specific component.

Internal brokers are described by the services they are *responsible* for providing (with their implementation), and the services they might request from other brokers. They also have a *state* consisting of typed instance variables which are (persistent) passive objects in the conventional sense of object-orientation. Such entities have an interface defined by a set of methods, do not provide services and cannot make service requests themselves. In addition to their state internal brokers can also have *sub-brokers* (i.e. other reactive components) to which they may *delegate* service requests. Such internal brokers which have sub-brokers are called *composite.* Brokers and their sub-brokers form a hierarchy with a predefined visibility of service requests. Internal brokers represent components explicitly developed to be reused as part of many CPEs. In our example the `ObjectManager` is an internal broker with a shared file repository as part of his state. The `check_out` method of the repository allows the checking-out of a file into the private workspace (state variable) of the `Editor`. The definition of `ObjectManager` is shown in Fig. 3.

```
BROKER ObjectManager: INTERNAL
STATE
   rep:Repository;
   result: OpResult;
   file: File;
RULES
   RULE CO1
   ON CheckOut(fn:FileName)
   DO result = rep.check_out(fn, file);
      reply(COResult, CheckOut, result, file);
```

Figure 3: Definition of the `ObjectManager` broker.

Interface brokers are also described by the services they are *responsible* for providing, and the services they might request from other brokers. They represent CPE-components used to enable the interaction of internal brokers and external tools. They encapsulate the functionality of an external tool such as in our case the `Editor`, the `UserInterface` and the `Compiler`. Interface brokers may also have a state consisting of typed state variables, but no sub-brokers. The `Editor`, for example, has as its state a private workspace for checked-out files. They provide a service interface corresponding to (part of) the functionality of the encapsulated component (e.g., `Display`) and translate service requests to the appropriate tool interface calls. They also permit the encapsulated component to use services provided by other components.

External brokers represent blackbox components whose implementation and state is not known but which participate in the process. The only information available about them, is which services such brokers may require from other components. They are used to model the behavior of human users and can request services from other external brokers as well as from interface brokers. In the example, the `SoftwareEngineer` is an external broker requesting edit and compilation services.

The characteristics of the three kinds of brokers are summarized in Table 1.

Broker type	sub-brokers	internal state	responsibilities	requests
internal	yes	yes	yes	yes
interface	no	yes	yes	yes
external	no	no	no	yes

Table 1: Characterization of broker types

3.4 Roles

Roles allow the specification of different responsibilities of a specific component in various situational and organizational contexts. Roles are defined as groups of component responsibilities and allow the modelling of the fact that the same physical entity (e.g. a person), may have different responsibilities in different organizational sub-groups. Broker definitions contain a set of role specifications. However, at any particular point in time a specific broker may assume only one role. Change of roles is dynamic, i.e. it can occur during the lifetime of a broker and is invoked by an internal operation (`change_role`) by the broker itself, as a reaction to an event or an explicit request. Each role specification is associated with a set of ECA-rules. ECA-rules describing behavior common to all roles of a broker (i.e. role-independent) may also exist.

In the context of a more complete process model example, roles may be used to differentiate the behavior of the person `SoftwareEngineer` as a programmer or a code-reviewer.

3.5 Broker Interaction Primitives

Interaction in the broker/services model involves at least two brokers, and is conceptually based on broadcasting. A broker initiates an interaction primitive which is visible to the other brokers in the system. There are a number of interaction primitives used for broker initialization, communication and synchronization. These interaction primitives are summarized in Table 2.

Operation	Parameters	Synch/ Asynch	Description
initialize	broker_name, initialization parameters	A	initialize a subbroker
request	service name, service parameters	S/A	request a service
reply	reply name, reply parameters	-	reply to a synchronous request
delegate	service name, service parameters	S	delegate a request to subbrokers
raise_exception	exception name	A	raise an exception

Table 2: Broker operations

PM enactment is done in brokers in response to various situations of interest represented by events (as described below). Such events include the signalling of interaction primitives such as *synchronous* or *asynchronous* service requests made by client brokers. In case of synchronous requests, the further processing at the client is blocked until a reply is provided. In case of asynchronous service requests, further processing takes place in the client after the request. Thus the semantics of synchronous requests correspond to that of dependent subactivities, while asynchronous requests trigger independent activities. Eventual results of such independent activities may be returned to the initial broker by another service request. A broker may choose to *delegate* a service request to his subbrokers instead of processing it itself. The sub-brokers must then return their results to their parent.

The broker interaction primitives contain system-provided (implicit) and user-provided mandatory and optional parameters. Implicit parameters to all interaction primitives include the name of the broker initiating the primitive and a unique primitive identifier used to distinguish between repeated calls. In case of replies the original request identifier is also included. Optional explicit parameters can be used in service requests, replies and delegation of services and are defined in the service signature.

Broker interaction primitives are used in our example for service requests (e.g. `Compile`, `CheckOut)` and replies (e.g. `Checkout OK`), for the initialization of the participating internal and interface brokers, for exceptional situations interrupting the normal process flow (e.g. if the display is not available — `DisplayUnavailable`), or for delegation of work to sub-brokers.

3.6 ECA-Rules

Brokers react to situations of interest represented by events. Their reaction to specified external events of various types is defined by *ECA-rules*. ECA-rules have a system-wide unique name and consist of an event clause, a condition clause, and an action part which is executed if the condition clause is satisfied. It is possible that more than one rule reacts to the same event within one broker. The execution of ECA-rules takes place within individual transactions. The processing of individual rules in a broker can be performed in parallel; ECA-rules thus define the unit of concurrency in our model.

In order to describe events occurring during the operation of a CPE, we use various *event types*. Our model uses the following *primitive event types*:

- *Broker interaction events* are explicitly raised by brokers through the (parametrized) interaction primitives introduced above.
- *Time events* occur when a particular point in time is reached. They are specified either absolutely (by giving a clock time), relatively to another event, or as periodic events.
- *Value events* are related to the modification of broker state variables. Such events are defined for update operations on the state variables and may occur immediately before or after the operation that updates the value of the variable.
- *Method events* are bound to the execution point of a method of a passive object in the broker state. Their occurrence point is specified as being just before or immediately after (i.e. directly before the method returns to its caller) method execution.

Note that the last two event types can only refer to persistent passive objects.

Monitoring intervals may be used to specify a — possibly implicitly defined — time interval in which an event has to occur in order to be considered as relevant. The monitoring intervals are an integral part of the event definition.

In order to adequately model broker behavior in more complex situations occurring during process execution we introduce *composite events*. Composite events are defined by (recursively) combining component events through the following constructors:

- *conjunction*: occurs when both component events have occurred,
- *disjunction*: occurs when one of the two components has occurred,
- *sequence*: occurs when the component events have occurred in the specified order,
- *negation* of an event: occurs when the component has not occurred within a specified time interval,
- *times*: occurs when the component event has occurred a specified number of times within a certain time interval,
- *closure*: occurs when the component event has occurred at least once within a specified time interval, but is signalled only once regardless how often the component event actually occurred.

Monitoring intervals are mandatory in the last three cases, and optional in the other ones.

Conditions in ECA-rules are queries expressed over the state of brokers or over the values of the event parameters and guard the execution of the action part. The condition is satisfied if queries evaluate to true or returns a non-empty answer. In the action part of the ECA-rules, various broker interaction operations (e.g. service requests and replies, delegation etc.), calls to methods of the broker state objects (e.g. database update operations), or signalling of events may be performed in order to implement services.

A partial ordering of rule execution can be defined by using a precedence clause. It is important to note that a precedence order has to be defined in case the action part of a rule affects the condition part of another one, and thus influences rule execution semantics.

The broker rules needed to model the example process are as follows:

- Rules of the `SoftwareEngineer` role `Programmer` requesting the editing and compilation services (MC1, MC2).
- `Editor` rules reacting to the initial edit request by requesting the check-out and display of the module file (Ed1, Ed2). A reply to the check-out request is required. The reply is handled by rules COR1 and COR2.

- An `ObjectManager` rule reacting to the check-out request (CO1). Note that this rule has to provide a reply to Ed1.
- `Compiler` rules to describe the compilation (C1) and the process after compilation. According to the results testing can start or the code has to be modified again (C2, C3).

4 Modeling CPEs with Brokers

The modeling of CPEs consists of the definition of its component architecture (i.e. its brokers) and the processes they can execute. During the modeling of the component architecture, the following activities are performed:
- processing entities (humans or software tools in the real world to be modeled) are identified and brokers are defined for them,
- if a processing entity can act in different contexts or positions, then appropriate roles are defined for the corresponding broker,
- tasks in the real-world are modeled through service definitions, and
- the fact that specific processing entities can perform activities is expressed through responsibilities.

Furthermore, within the CPE-model further brokers are defined, providing functionality not provided by external tools i.e., services such as object management, presentation services etc. The structural modeling process is inherently iterative.

The PM is defined by the following steps:
- the process structure is defined in terms of steps (activities performed by brokers),
- execution dependencies between the various steps are modeled.

The execution dependencies can be of three different kinds [26]:
1. *execution state dependencies* constrain the execution order of steps according to the state of other steps,
2. *output value dependencies* specify constraints on relationships between input/output data of two or more steps, and
3. *temporal dependencies* specify time constraints on steps, such as deadlines.

Simple dependencies can be combined to more complex ones through constructors such as AND-JOIN (e.g., multiple steps must all have been finished before the dependent step can begin execution).

In Fig. 4 the service requests and replies of the example process are shown. Their sequence is determined by rule conditions. The complete rule set is shown in the appendix.

5 Broker Implementation

In this section, we describe how the broker/services model is currently implemented. This includes a *platform* that is used for each constructed CPE, and the implementation of brokers on top of this platform. An important objective concerning the choice of the platform is to minimize the overall implementation effort. Concerning the platform, we have the following requirements:
- since processes are long-lasting and the accessed data items and process state information are persistent, the platform must have database functionality,
- since we use object-orientation for “passive” data, the platform should be object-oriented as well, and
- the CPE-platform should be able to monitor and control process executions and to detect the relevant situations of interest.

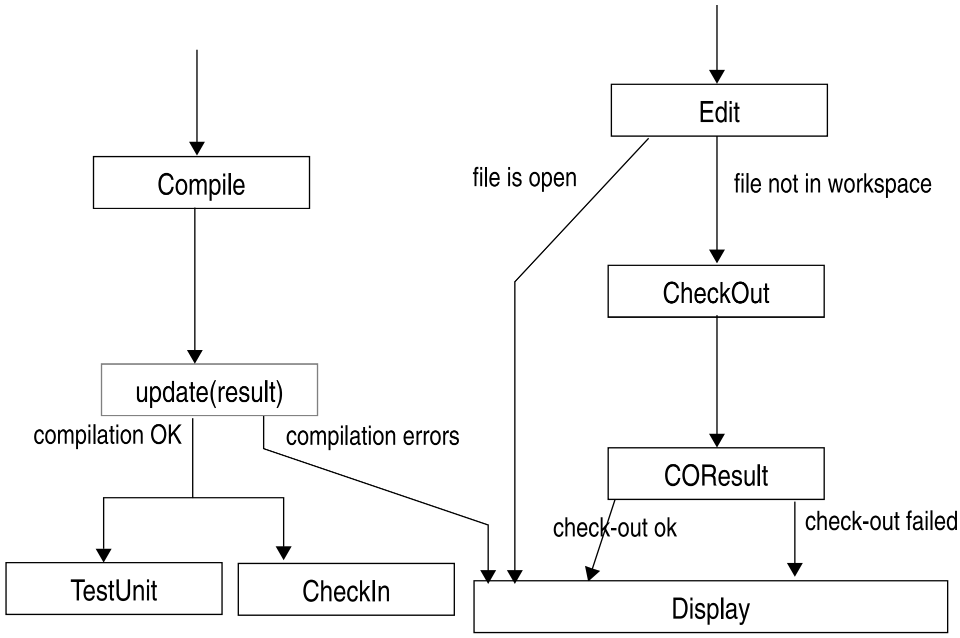

Figure 4: The dynamic structure of the example process. Request and reply events are represented as solid boxes, other events as dotted boxes, while the possible request sequence is determined by rule conditions.

The above requirements are best satisfied by an active, object-oriented database management system (ADBMS) [9]. Currently, we use our home-grown ADBMS SAMOS [17] which is based on the commercial object-oriented DBMS ObjectStore [21].

5.1 The Use of the Active DBMS SAMOS as a CPE-Platform

Much of the functionality of the CPE-framework is provided directly by the capabilities of SAMOS. In addition to passive data modeling facilities, SAMOS supports the specification (and implementation) of reactive behavior by means of ECA-rules (henceforth called SAMOS ECA-rules). Events in SAMOS can be primitive or composite. Primitive events can in turn be of different kinds: *message sending event* (occurs at the beginning or the end of a method execution), *value event* (occurs before or after the value of an object is modified), *transaction event* (occurs before or after a transaction operation), *time event* (occurs at a specific point in time, periodically after a specified interval, or as soon as a specified time interval following another event occurrence has elapsed), and *abstract event* ("occurs" when explicitly signalled by a user or application). Furthermore, SAMOS allows the definition and detection of composite events specified with the following event constructors: *conjunction*, *disjunction*, *negation*, *sequence*, closure, and *times*. For a definition of the semantics of these constructors, see [15].

SAMOS is used to provide event management and notification, persistent storage and transaction support. Brokers are implemented as SAMOS/Objectstore objects. Broker definitions are translated into instances of classes derived from the BS++ class hierarchy we are currently developing. BS++ provides an application framework containing base classes for the three broker types, broker roles, passive components, broker interaction primitives and other event types, and rules describing broker behavior. Broker ECA-rules are transformed into SAMOS ECA-rules. In the rest of this section, we will concentrate on how the underlying active DBMS provides run-time support for brokers. Other aspects of the implementation of brokers are described in [18].

5.2 Broker Interaction and Communication

The integration of CPE-components is similar to implicit invocation [13]. Brokers provide a pure event interface and not a method interface as well to other brokers in the system. Passive objects in the broker state on the other side, provide a conventional method interface which is however visible to their parents only. Autonomous system components (i.e. brokers) communicate thus only through (parametrized) events, so that no broker knows in advance which other broker will be affected by these events. The advantage of this approach with respect to process modeling and enactment in CPEs is that brokers can be added to or deleted from the system easily, evolving the process dynamically.

Brokers are translated into instances of SAMOS classes. SAMOS is responsible for signaling simple and composite events to objects which have registered their interest in the occurrence of such events (e.g., brokers that have ECA-rules defined containing that event). Broker interaction primitives are implemented as parametrized *abstract* events which have to be explicitly signalled by an application. Time, value, and method events are implemented based on their SAMOS counterparts.

When a simple event occurrence is signalled (e.g., by the execution of the action part of a broker ECA-rule or by the arrival of a specific point in time), it is recognized by SAMOS which is then responsible for finding all registered (simple and composite) event definitions that contain this event. The rules containing patterns of events which have occurred can then be executed. The underlying ADBMS is thus acting as a message server broadcasting event occurrences to all interested brokers.

5.3 Transaction Support

As mentioned in the requirements section, transaction support should be provided for individual steps of an executing process, i.e., single steps of a process should be atomic and isolated from other activities in the running CPE. Concurrency between steps of a process or between different processes should however be possible, unless prohibited by dependencies between the steps. Thus, the first problem is how to map actions of brokers into appropriate transaction structures of SAMOS. Furthermore, some transaction properties are desired for entire processes, too. Intra- as well as inter-process concurrency should be allowed in a controlled manner. Second, processes should be recoverable (i.e., it should be possible to abort processes and to compensate steps, and processes should be protected against system crash or media failure). However, processes are different to classical transactions in several respects. They are long-lasting activities, and the notion of correctness and recoverability are typically process-model-specific. Processes are thus not feasible units of atomicity. Subsequently, we describe the mapping of process activities into transactions of the next lower level (similar to the approach described in [8]).

In our model, each ECA-rule represents an individual process step and is executed under transaction protection. This is the case with internal and interface brokers which contain state variables accessed by conditions and/or actions of their rules. Events may trigger more than one rule (possibly in more than one broker) which then might be executed concurrently. Thus the access to the state variables of brokers by their ECA-rules must be controlled by transactions. The problem also exists with external brokers which do not have state variables but access the optional parameters used by broker interaction operations.

Our model is implemented using the execution model of SAMOS, which in turns uses ObjectStore's nested transaction model [23]. Transaction structures of the SAMOS/Objectstore implementation are determined based on the broker operations as follows:

1. A broker action can be the sending of a message to one of its passive objects. In this case, a subtransaction is started. In this way, operations on passive objects are always abort-dependent on the transaction which contained the message sending.
2. A broker action can be a broker interaction primitive, such as `request`. We distinguish the following cases depending on the specified synchronization mode:
   a. If the synchronization mode is synchronous, then the called operation “interrupts” the calling operation and is considered to be dependent on the calling one. These characteristics are achieved through executing the operation in a subtransaction.
   b. If the synchronization mode is asynchronous, then the calling and the started transaction can in principle be executed concurrently, and no dependencies between them are established. Thus, a new root transaction is started.

For establishing these transaction structures appropriately, we use the various *coupling modes* [20] between the event, condition, and action parts of SAMOS ECA-rules as described below. They define the resulting transaction structures of rule execution as well as the point in time when the rules are executed. With immediate coupling, the triggered rule is executed immediately in a subtransaction and interrupts the triggering transaction. If the decoupled mode is specified for a rule, then the rule is executed in a separate root transaction (possibly concurrently to the triggering transaction).

Summarizing, the implementation ensures that broker actions (and atomic steps) are failure-atomic and isolated. Since SAMOS objects (at least currently) cannot be internally concurrent, intra-process parallelism in brokers is restricted. Also, transaction management of entire processes is still an open issue and left to future work (i.e., modeling processes as long-lasting, nested transactions).

6 Related Work and Concepts

Our work has benefited from various research domains such as concurrent programming languages [3], agent-oriented programming [28], and software architecture [14]. In this section we briefly survey related work from these fields.

Concurrent Object-Oriented Programming Languages. *Actors* [1] describe a model of extremely fine grained concurrency based on autonomous agents which is not optimized for modeling real-world entities. Their communication model is based on asynchronous message passing with non-blocking sends. HAL [2] is a high-level actor language with a lisp-like syntax which is compiled to C code. HAL extends the actor communication model by providing a message order preserving send and synchronous RPC sends. In contrast to brokers, processing in actor systems is based on processing incoming messages addressed to a specific actor. In order to handle incoming messages which cannot be processed, a special forwarding construct has to be provided. Similarly to brokers, actors have an identity and a current behavior defining how the actor will react to the next incoming message, the existence of an actor however depends on other actors knowing its address.

The Oz Programming Model [29] extends the basic concurrent, non-parallel constraint model and is a generalization of functional and object-oriented programming. The computation space in Oz consists of tasks connected to a blackboard and synchronized by it. The high-level modelling constructs provided in Oz are procedures, objects (representing concurrent data structures with encapsulated state and modelled as procedures applied to messages) and agents (computational abstractions processing messages received through ports). Agents are considered active objects. Oz has no built-in support for persistence and compared to our model, emphasizes more on low-level implementation aspects.

Another concurrent object-oriented programming language for active objects is Hybrid [24]. *Domains* define the units of concurrency in Hybrid. They are collections of active objects (roughly corresponding to composite brokers in our model) executing in one process. At any given time, however, only one object may be active in a domain limiting the usefulness of active objects for modeling CPE-components. Objects communicate with independent objects through message passing.

Agent-Oriented Programming. A natural extension of concepts defined in concurrent object-oriented programming language research [3] is the notion of *agents*. Agents are autonomous components encapsulating an internal state and communicating with other agents through messages in some kind of agent communication language. They have the ability to react to changes in their environment and are pro-active in the sense of demonstrating a goal-oriented behavior. Brokers are also reactive, autonomous components with an internal state and interact with other brokers through the described primitives. Their behavior is however governed by a PM expressed by their ECA-rules and as such cannot be regarded as goal-oriented. The concept of an agent in AI as exemplified by *agent-oriented programming* [28] is used as a specialization of object-oriented programming, and defines additional criteria for *agenthood* by fixing the mental state of an agent which now consists of beliefs, capabilities, and decisions. It also constrains the types of allowed messages and agent methods.

Software Process Modeling. Many approaches have been presented in the literature for the modeling and organization of software processes and other long-lasting cooperative activities in process-centered software development environments [e.g., 10]. A survey of process modeling formalisms is beyond the scope of this paper. However, these can be broadly categorized into procedural, petri-net or automata-based, trigger-based and rule-based. In most research prototypes, a hybrid approach to process modeling such as our own is used, as there seems to be consensus among researchers that not all aspects of a PM can be adequately represented by only one formalism. Compared to other research, our approach emphasizes the aspects of integration and modularization of component and process architecture with the objectives of achieving efficient process evolution and high component reusability.

Rule-based concurrency control and coordination in cooperative environments has been addressed by several researchers [e.g. 4, 6, 19]. The focus of our work which uses concepts such as those presented in [6] emphasizes the provision of a complete modeling formalism and a programing environment for the construction of CPE frameworks using the functionality provided by an active DBMS.

Composition-Based Approaches. Examples of research efforts concentrating on the development of a framework for constructing process-centered software development environments are Oikos [22] and ALF [6]. In Oikos enactable PMs consist of a hierarchy of interacting reactive systems which belong to predefined classes describing entities encountered in the software development domain. The entities can be combined in ways predefined for their respective class and their interaction is based on the client-server model. An integration framework based on Unix processes communicating through sockets provides runtime support. In ALF a metamodel for the description of software production processes has been developed. The metamodel provides a structurally object-oriented description of production data, operators with pre- and postconditions to express the execution order of activities, and ECA-rules for the description of process steps whereby the supported event types only include database operations.

Finally, research is also interested in the construction of general, open concurrent object systems based on composition of reusable artifacts into coherent systems [25]. While in this area no widely agreed-upon results have been found, composition and reusability are also long term goals of our work.

7 Conclusion

In this paper, we have presented the broker/services approach to modeling and implementing process-oriented environments on top of an ADBMS. Two major questions are still open and subjects of future work:

- a complete transaction management concept for CPEs,
- a complete "programming environment" for brokers and services.

Currently, transaction management is provided only for the steps of a process, but not for entire processes. Thus, the first (partially) open issue is transaction management. We anticipate however, that functionality such as recovery, cooperation, and concurrency control will typically be process specific (e.g., compensation of specific steps in the process upon abort of single steps or entire processes).

So far, brokers and services provide abstractions used during the development of a CPE. In spite of the usefulness of these abstractions, we do not want to "program" brokers and construct CPEs from scratch all the time, so more support for the construction process is necessary. Hence, a CPE-development environment is desired that allows a more declarative modeling of processes. This environment should support the reuse of CPE-artifacts (such as PMs, broker designs, etc.). Our objective is an appropriate composition language for CPEs in the sense of [25]. Furthermore, while the architectural style of implicit invocation has strong advantages, it also has several disadvantages (e.g., understanding the behavior/semantics of a CPE). We thus will investigate more formal models for brokers, and tools for simulating PMs and visualizing process states.

8 References

1. G. Agha: *Actors: A Model for Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
2. G. Agha, C. Houck: Hal: A High-level Actor Language and its Distributed Implementation. *Proc. 21st Intl. Conf. on Parallel Processing* (ICPP), St. Charles, August 1992.
3. G. Agha, P. Wegner, A. Yozenawa (eds.): *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, Cambridge, MA, 1993.
4. J.-M. Andreoli, H. Gallaire, R. Pareschi: Rule-based Object Coordination. In [5].
5. P. Ciancarini, O. Nierstrasz, A. Yonezawa (eds): *Object-Based Models and Languages for Concurrent Systems*. ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, July 1994. LNCS 924, Springer 1995.
6. G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, J. Longchamp: ALF: A Framework for Building Process-Centered Software Engineering Environments. In [10].
7. R.G.G. Cattell: *Object Data Management*. Revised Edition, Addison Wesley, 1994.
8. U. Dayal, M. Hsu, R. Ladin: Organizing Long-Running Activities with Triggers and Transactions. *Proc. ACM SIGMOD*, Atlantic City, NJ, May 1990.
9. K.R. Dittrich, S. Gatziu, A. Geppert (eds): The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In [27].
10. A. Finkelstein, J. Kramer, B. Nuseibeh (eds): *Software Process Modeling and Technology.* Research Studies Press Limited, 1994.

11. S. Frolund, G. Agha: A Language Framework for Multi-Object Coordination. In O.M. Nierstrasz (ed): *Proc. 7th European Conf. on Object-Oriented Programming* (ECOOP), Kaiserslautern, Germany, July 1993. LNCS 707, Springer.
12. H. Garcia-Molina, K. Salem: Sagas. *Proc. ACM SIGMOD*, San Francisco, May 1987.
13. D. Garlan, D. Notkin: Formalizing Design Spaces: Implicit Invocation Mechanisms. *Proc. VDM '91: Formal Software Development Methods*, Noordwijkerhout, The Netherlands, October 1991. LNCS 551, Springer 1991.
14. D. Garlan, M. Shaw: An Introduction to Software Architecture. In V. Ambriola, G. Tortora (eds): *Advances in Software Engineering and Software Engineering, Vol. II*. World Scientific Publishing Company, 1993.
15. S. Gatziu, K.R. Dittrich: Events in an Active Object-Oriented Database System. In N.W. Paton, H.W. Williams (eds): *Rules in Database Systems*. Springer-Verlag, 1994.
16. D. Georgakopoulos, M. Hornick, A. Sheth: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* 3:2, Kluwer Academic Publishers, September 1994.
17. A. Geppert, S. Gatziu, K.R. Dittrich: Architecture and Implementation of an Active Object-Oriented Database Management System: the Layered Approach. Technical Report, Institut für Informatik, Universität Zürich, 1995.
18. A. Geppert, M. Kradolfer, D. Tombros: Realization of Cooperative Agents Using an Active Object-Oriented Database Management System. In [27].
19. G. Heineman: Process modeling with Cooperative Agents. In B. C. Warboys (ed.): *Proc. EWSPT'94: Software Process Technology,* Villard de Lans, France, February 1994. LNCS, Springer 1994.
20. M. Hsu, R. Ladin, D.R. McCarthy: An Execution Model for Active Database Management Systems. *Proc. 3rd Intl. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.
21. C. Lamb, G. Landis, J. Orenstein, D. Weinreb: The ObjectStore Database System. *Communications of the ACM* 34:10, 1991.
22. C. Montangero, V. Ambriola: Oikos: Constructing Process-Centered SDEs. In [10].
23. J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
24. O. Nierstrasz: Active Objects in Hybrid. *Proc. Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Orlando, FL, October 1987.
25. O. Nierstrasz, T.D. Meijler: Requirements for a Composition Language. In [5].
26. M. Rusinkiewicz, A. Sheth: Specification and Execution of Transactional Workflows. In W. Kim (ed): *Modern Database Systems*. ACM Press / Addison Wesley, 1995.
27. T. Sellis (ed): *Proc. 2nd Intl. Workshop on Rules in Database Systems*, Athens, Greece, September 1995. LNCS, Springer 1995.
28. Y. Shoham: Agent-oriented Programming. *Artificial Intelligence* 60, March 1993.
29. G. Smolka: The Oz Programming Model. To appear in J. van Leeuwen (ed.): *Computer Science Today*, LNCS, Springer 1995.
30. C. Tomlinson, M. Scheevel: Concurrent Object-Oriented Programming Languages. In W. Kim, F.H. Lochovsky (eds): *Object-Oriented Concepts, Databases, and Applications*. ACM Press, New York 1989.

Appendix: The Complete Example

```
BROKER SoftwareEngineer: EXTERNAL
ROLE Programmer
RULES
   RULE MC1                          RULE MC2
   ON  ...                           ON ...
   DO  request(Edit, fn, ASYNC);     DO request(Compile, fn, ASYNC);
//------------------------------------------------------------------------
BROKER Compiler: INTERFACE
STATE
   result: CResult;
RULES
   RULE C1
   ON  Compile(fn: FileName)
   DO  result = compile(fn);

   RULE C2
   ON  update(result)
   IF  result.cstatus == OK
   DO  request(CheckIn, file, ASYNC);
       request(TestUnit, result, ASYNC);

   RULE C3
   ON  update(result)
   IF  result.cstatus == NOK
   DO  request(Display, result, ASYNC);
//------------------------------------------------------------------------
BROKER Editor: INTERFACE
STATE
   ws: Workspace;
   result: OpResult;
RULES
    RULE Ed1                              RULE Ed2
    ON   Edit(fn: FileName)               ON  Edit(fn: FileName)
    IF   ws.contains(fn) == FALSE         IF  ws.contains(fn) == TRUE
    DO   request(CheckOut, fn, SYNC);     DO  request(Display, file, ASYNC);

    RULE COR1
    ON   COResult(result:OpResult, file:File)
    IF   result.status == COOK
    DO   request(Display, file, ASYNC)

    RULE COR2
    ON   COResult(result:OpResult, file:File)
    IF   result.status == COFail
    DO   request(Display, result, ASYNC);
//------------------------------------------------------------------------
BROKER ObjectManager: INTERNAL
STATE
   rep:Repository;
   result: OpResult;
   file: File;
RULES
   RULE CO1
   ON  CheckOut(fn:FileName)
   DO  result = rep.check_out(fn, file);
       reply(COResult, CheckOut, result, file);

   RULE CO2
   ON  CheckIn(file:File)
   DO  rep.check_in(file);
```