In T. Sellis (ed.): Proc. 2nd Workshop on Rules in Databases (RIDS), Athens, Greece, September 1995. Lecture Notes in Computer Science, Springer 1995.

Realization of Cooperative Agents Using an Active Object-Oriented Database Management System

Andreas Geppert   Markus Kradolfer   Dimitrios Tombros

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

Abstract: Cooperative, process-oriented environments (CPEs) are systems whose behavior is defined in terms of process models. We show how CPEs are realized through brokers, which are a special form of software agents[1] used to model participating entities in CPEs. A broker can represent a human participant, an existing software tool, or a part of the environment infrastructure. In our approach, we implement brokers on top of the active object-oriented database management system (ADBMS) SAMOS. Particularly, we use the facilities of SAMOS for implementing communication/cooperation between and control of brokers in CPEs. Our approach allows the construction of flexible, extensible systems and the definition of the behavior of participating entities local to brokers, avoiding the need for a centralized process engine.

Keywords: active database systems, software processes, workflows, agent control

1 Introduction and Motivation

Research in cooperative process-oriented environments is currently a very active topic. Example types of CPEs are process-centered software development environments (PCDEs) [9] and workflow management systems (WFMSs) [13, 23]. Regardless of whether software processes can actually be treated as a special case of workflows, as for example stated in [19], PCDEs and WFMSs have in common the notion of process. Both support the computer-based modeling and execution of processes[2], whereby a process has the following characteristics:

- it is a possibly long-lasting activity,
- it consists of several sub-activities (steps or tasks),
- the execution of these sub-activities can be constrained (e.g., through execution order, predicates on input or output data, or timing constraints), and
- it may involve both human beings and tools.

Multiple processes may have to be modeled and executed in the same environment, and different CPEs may comprise different component systems (e.g., tools). Thus, it is not feasible to define one concrete, fixed CPE. Instead, a CPE framework is required

1. We use the term “agent” in the sense of “processing entity” as it is used in the workflow literature [e.g., 6]. Brokers represent software agents [12]. Both terms should not be confused with the concept as it is used in AI.
2. Subsequently, the term “process” subsumes “workflow” and “software process”.

that can be customized to specific requirements. Such a framework has to satisfy the following criteria:

- it should support modeling of the structure and execution semantics of processes,
- it should support the execution (enactment) of processes and thus:
  - provide *communication* between agents,
  - *coordinate* the various agents (humans and — possibly external — tools) according to the process model, and
  - *control* the process state and progress. Depending on the process state (and the desired process semantics), the CPE must react appropriately.

Most PCDEs or WFMSs support coordination and control through a *process engine* (synonymously, task manager, activity manager). This component is central to the system and guides processes, while most other components (agents) are more or less passive, i.e., they can act only as far as allowed or explicitly requested by the process engine. The problem with such a process engine is its complexity, especially if *process evolution* should be supported, too. The complexity of a centralized process engine stems from the fact that it has to keep all information on the possibly dynamically changing capabilities of participating agents and on the state of process execution. Moreover, process engines do hardly support an architectural view integrating services, software agents, and process logic.

We propose *brokers* and *services* as constructs for building CPEs. They provide a service-oriented view of the environment and are used to model both, the static architecture and the behavioral aspects of CPEs (i.e., the process model). Brokers are able to detect situations in which they have to react automatically, so that control and coordination can be decentralized and distributed among the components of the CPE. Such situations are often more complex than simple service requests. For instance, a broker must be able to realize the fact that two alternative tasks have both failed, or that a specific deadline is only one week away, and so on. We show that such powerful brokers can be implemented on top of an advanced ADBMS supporting composite events. The advantage of using this implementation platform is that we can keep the description of agent behavior and process control information local to the participating agents instead of having a centralized process engine interpreting a process program responsible for the tasks of agent coordination and cooperation.

The remainder of this paper is organized as follows. In the next section, we describe the broker/services model. In section 3 we show how brokers and services are used for the realization of CPEs. Section 4 introduces the ADBMS SAMOS and shows how it is used to implement brokers. Section 5 presents an example, section 6 briefly surveys related work, and section 7 concludes the paper.

2 The Broker/Services Model

In this section, we introduce the *broker/services model* we use to describe the structure of a system, its behavior, and architectural constraints. By "system" we mean an environment that consists of several agents (applications, environment infrastructure components, human users). In order to obtain an integrated architecture, these agents are represented by *brokers*. Due to reasons which become apparent later, the ADBMS SAMOS [10, 16] is always part of the environment infrastructure. We further assume that agents offer *services* to their potential clients, and that the system behavior is defined by a *process model* determining under which circumstances a specific service can or must be provided.

Our model uses an object-oriented approach to system construction, extended with the possibility to define reactive behavior of the participating objects[3]. In addition to the concepts introduced above, we use *responsibilities* to relate services with the broker(s) responsible for their provision. A CPE-architecture is then defined as a collection of brokers operating in various roles, responsible for providing services and able to monitor complex events and react according to predefined ways. The services provided by brokers can refer to the manipulation of data or to the control and coordination of other brokers. Below, we describe these concepts in more detail. Fig. 1 contains a textual specification of the key concepts of the model. Note that the process model will be encoded within broker definitions; an example thereof is presented in section 5.

2.1 Services

Services model the functionality of system components. The totality of services provided by components represents the functionality of the entire system. The use of services allows a view of the environment abstracting from concrete implementations. A specific service is provided by one or more brokers (see below) and can be requested by various client brokers. It is specified by a service signature consisting of the service name, its parameters, the possible replies and exceptions its request may cause.

2.2 Brokers

Brokers represent agents, responsible for the provision of system services. In order to model varying degrees of integration in the CPE, we distinguish between three kinds of brokers: *internal*, *external,* and *interface* brokers. Internal and interface brokers are described by their state, the services they are *responsible* for providing, and their reaction to predefined events. External brokers represent agents whose "implementation"

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

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

RESPONSIBILITY service_name: broker_name
```

Figure 1. Key Concepts of the Broker/Services Model .

3. The term "reactive behavior" describes the capability of objects to autonomously execute various actions in response to the occurrence of predefined situations (not just method calls).

is not known (e.g., humans). The state of a broker consists of typed instance variables which can be either *sub-brokers* or passive objects. Passive objects can be used to represent data manipulated by a broker, thus providing a mapping to a common data model for all participating agents. Brokers which have sub-brokers are called *composite.* Sub-brokers of internal and interface brokers can only be internal or interface brokers themselves.

Internal brokers represent proprietary CPE components. Interface brokers implement the behavior of proprietary CPE components interacting with human beings and external tools. A typical example is a session manager representing the human user interface to the system. External brokers are blackboxes for which the internal state and service implementations do not have to (but can) be defined. They model the behavior of human users and external tools, and can request services from other external brokers as well as from interface brokers.

Brokers and their sub-brokers form a hierarchy with a predefined visibility of service requests. This allows the definition of different behavior according to the organizational context of a request.

2.3 Roles

Roles specify the responsibilities of brokers in various situational and organizational contexts. The concept of roles is used to model for example the fact that the same agent (e.g. a person), may have different responsibilities in different organizational sub-groups. Roles are used in a slightly different sense to the conventional in workflow modeling, where they define a grouping of capabilities [6] usually associated with a functional objective in an organization. An example of this definition of a role is to "be a manager" and thus every person who is a manager plays this role. In our case however, roles denote the responsibilities a concrete broker (human or non-human) has in a (sub)organization at some point during its lifetime. Thus roles can only be defined as part of a broker definition. Each role specification consists of a set of event-condition-action-rules (ECA-rules) and state variables. There may however be state variables and ECA-rules common to all roles of a broker (i.e. role-independent).

2.4 ECA-Rules

ECA-rules define the reaction of brokers (within the context of a role) to specified external events of various types. They have a unique name and consist of an event clause, a condition clause, and an action part. It is possible that more than one rule reacts to the same event within one broker (role).

The events to which brokers react can occur for example due to a sequence of broker actions within a process, or when specific points in time are reached. 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*:

- *Service provision events* are explicitly raised by brokers through special (parametrized) operations. These include the events generated by service requests and their subsequent replies. The events have parameters corresponding to those needed by the specific operations.

- *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 an object value. This allows among others the monitoring of (database) object states. Such events are defined for update operations on object attributes and take place before or after the operation that updates the value of the object is performed.
- *Method events* are bound to the execution point of a specific passive object method. 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 passive (database) objects.

Time intervals can be defined in order to limit the period in which an event occurrence is of interest and should be monitored. Such *monitoring intervals* 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 a part of the event definition.

In order to adequately model reactive broker behavior in more complex situations (e.g. within the context of process control, see below) we introduce *composite events*. Composite events are defined by combining component events — possibly recursively — 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 event 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.

While monitoring intervals are mandatory in the last three cases, they are optional in the first three ones.

Conditions are expressed over the state of brokers and guard the execution of the action part. In the action part of the broker role ECA-rules, various operations (e.g. service requests and replies), or calls to methods of broker-specific passive components may be performed in order to implement services. Due to space economy, these are only exemplified in section 5.

A partial ordering of rule execution can be defined by using a precedence clause. A precedence order has to be defined in case the action part of a rule affects the condition part of another one, therefore influencing rule execution semantics.

3 Modeling Process-Oriented Environments With Brokers

3.1 Requirements

A CPE framework should meet the following requirements:

- it should be customizable in an *abstract* and *declarative* way, to the functional and operational requirements of specific organizations and projects using the developed CPE,
- it should support different, but *integrated views* of the functionality it offers,
- it should support *communication* between participating agents of the various sorts,
- it should support *coordination* between participating agents of the various sorts,
- it should be able to *control* participating agents wherever possible and necessary.

Naturally, the process-oriented view describing tasks, their structure, and related constraints is important in a CPE. The process-oriented view describes the "process logic" since it defines the "how" of a process model. We additionally require that a CPE provides activity-oriented and agent-oriented views as well. The first one focuses on the activities performed by the CPE agents and the services used for the realization of the CPE functionality, while the latter one additionally supports the assignment of tasks to concrete agents.

An integrated, abstract view is thus needed. Generally, it can be the case that CPE components are implemented on different platforms using even different data stores. It must still be possible to have a level providing a uniform view of the entire CPE, i.e., heterogeneous component systems should be integrated into a coherent environment. Given that components may be heterogeneous but still have to interoperate, communication cannot simply be realized through message passing. Appropriate mechanisms for communication between agents have to be provided at a higher level of abstraction.

The same holds for coordination: it will seldom be the case that agents are completely independent from each other. Usually some of them will have to cooperate to various degrees in order to fulfill the overall task. In other words, the CPE should provide a mechanism that allows agents to be coordinated according to the process model.

Ultimately, agents must be controlled during process execution. The CPE must provide a mechanism for the enforcement of required constraints, the prevention of inconsistent process states and transitions, and the automatic reaction to such situations.

The first steps in customizing a CPE are the following:

- identify the processes to be modeled,
- determine the required tasks,
- determine agents that are responsible for specific tasks in one of the processes.

Clearly, a methodical approach for these steps is required. Nevertheless, we assume that this analysis has already been performed, and subsequently describe how brokers are used for the customization of a CPE.

3.2 Integrated Software Architecture and Different Views

Given the three steps mentioned above, the next step consists of determining the appropriate (static) software architecture. By software architecture, we mean

- a collection of brokers representing agents or internal components,
- a set of services, where each service either represents a task of a process or an internal service,
- a set of responsibilities assigning services to brokers.

Any agent in a CPE — be it a proprietary component, an external tool, or a human — is represented by a broker. On its top level, the CPE is integrated since all brokers interact via service requests and replies. Depending on the concrete agent, we typically know more or less about how the corresponding broker implements its services. For a proprietary component, we will know all implementation details. For external tools, we know their interface and the (operational) semantics of their operations. For humans, we know their responsibilities but do not (need to) have precise knowledge of how they do their work. This variety is captured through the specialization of brokers into the three subclasses of brokers mentioned above (internal, external, and interface brokers).

As mentioned before internal brokers represent proprietary components and interface brokers are used to represent external tools by providing a tool wrapping mechanism. The service implementation is actually a shell mapping service requests to the interface of the tool, collecting the tool output, and eventually returning the results. External brokers represent blackbox agents such as humans for which only service signatures are known.

3.3 Modeling Process Control

Control means enforcing constraints, including task dependencies and constraints on data items accessed and manipulated by some task of the process. The following kinds of task dependencies can be distinguished [21]:

- execution dependencies,
- data or value dependencies, and
- temporal dependencies.

Execution dependencies are defined through execution states of tasks. An execution dependency can for example state that upon termination of a task, another one has to be started. Data or value dependencies of tasks are expressed through the output values of other tasks or values which are accessed by arbitrary systems. A data or value dependency can for example state that task A has to be executed if task B terminates with an output value below a certain threshold. Temporal dependencies define arbitrary timing constraints on tasks like “task B has to be started within 6 weeks after the termination of task A”.

In the broker/services model, we can model the above dependencies by using ECA-rules. We can additionally model their combinations by using logical operators like conjunction, disjunction and negation on events. For example the execution dependencies “and-join” (e.g. only after a set of tasks has terminated, another (set of) task(s) must be started) and “or-join” (e.g. only when a certain number out of multiple (parallel) tasks has been terminated (successfully), another one can or must be started) described in [23] can be defined.

For example, assume that an execution dependency refers to previously executed tasks T_i ($1 \leq i \leq n$) whose termination is indicated through raising reply events including an optional list of parameters. Let the dependency require task T to be executed. Then the mapping of dependencies to composite events is as follows:

- an and-join is mapped to an event conjunction ($\wedge$)
- an or-join is mapped to an event disjunction ($\vee$),
- deadline dependencies are mapped to time events or events constrained by monitoring intervals.

Furthermore value or data dependencies are mapped to a parametrized event and a parameter test in a condition. Examples of these mappings are presented in Table 1.

Dependency	ECA-Rule defined for broker		
	Event	Condition	Action
data or value	reply of T1, including parameter p	if p	request T
execution (sequence)	reply of T_i	-	request T
execution (and-join)	$\wedge_i$ reply of T_i	-	request T
execution (or-join)	$\vee_i$ reply of T_i	-	request T
temporal (deadline)	! ($\wedge_i$ reply of T_i) within interval	-	notification etc.

Table 1. Examples of ECA-Rules for Modeling Task Dependencies

Current PCDEs or WFMSs typically implement control in some kind of process engine, which then keeps track of the process state. Some also use ADBMSs for control [e.g., 6, 17], which however are typically not able to detect complex situations (and therefore the process engine is nevertheless needed).

In our model, control can be completely performed by brokers on top of the ADBMS. Thus, control information is distributed among brokers and less centralized than in current systems. In addition, localized control leads to a more rigorous client/server approach. Apparently, some of the process engine's task are pushed into the ADBMS, which we see as an advantage in terms of using standard base components wherever possible and striving for a "minimality of concepts".

4 Implementation of Brokers

In this section, we introduce SAMOS and then show how it is used for the implementation of brokers.

4.1 SAMOS, an Active Object-Oriented DBMS

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 can be primitive or composite. Primitive events can in turn be of one of the following 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 (begin, commit, or abort transaction),
- *time event*: occurs at a specific point in time (absolute time event), periodically after a specified interval (periodical time events), or as soon as a specified time interval following another event occurrence has elapsed (relative time events), and
- *abstract event*: "occurs" when explicitly signalled by a user or application.

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 [11].

Upon event detection, the condition is checked. If it holds, the action is executed, otherwise the execution of the rule terminates. Both conditions and actions must be given in the data manipulation language (DML) of the underlying ooDBMS ObjectStore. For details of rule execution, which are less relevant in this context, see [16].

4.2 Implementation of Brokers Using SAMOS

For the implementation of brokers we strive to map the concepts present in the brokers to the functionality provided by SAMOS. We avoid however the extension of SAMOS functionality in order to abstract from a specific implementation platform. The concepts of interest in the mapping process are brokers and their responsibilities, passive broker components, services, replies and ECA-rules describing the reactive broker behavior. The underlying ADBMS (in this case SAMOS) is used to manage and detect events relevant to the brokers, to manage the rules describing broker behavior and to implement coordination and communication mechanisms for individual brokers.

Similar to SAMOS' approach to represent events and rules as objects [11, 16], we model brokers as instances of a class `broker` (Figure 2). Each broker contains a non-empty set of `role` objects. Among the methods of the `role` class are predefined operations with which brokers in a role can request services from other brokers (at the same level or subbrokers) or reply to incoming service requests. References to the rules describing the broker behavior in a role are also stored in order for example to locate the relevant rules when this behavior is modified.

Passive broker components can be of different types and are instances of children of a generic `component` class. They are referenced in an attribute of each broker instance (`Set<Comp*>`) and are declared such that brokers can call their methods.

During the mapping process ECA-rules defining the reaction of a broker `b` in a role `r` to a service request `s` are transformed to SAMOS ECA-rules as described below. Suppose we have two brokers `b1` and `b2` with `b1` being responsible to provide `s1`

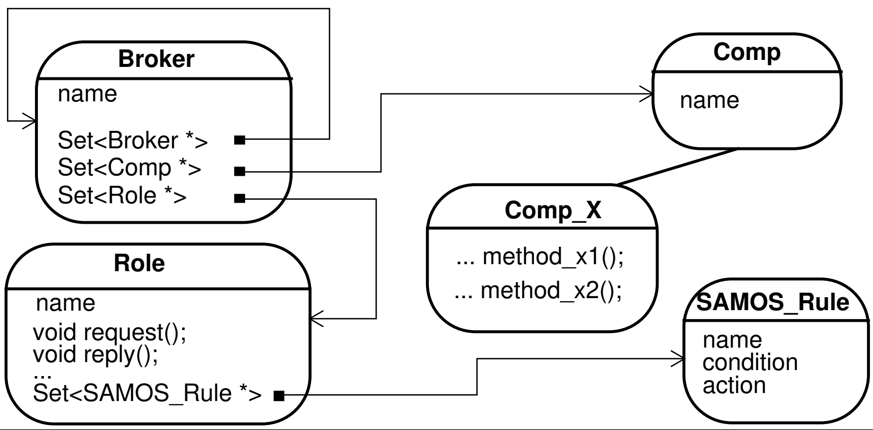

Figure 2. Brokers and components

when in the role `r1`. Suppose further that in order to provide `s1`, `b1` requests the service `s2` provided by `b2` (in role `r5`) by sending a message `m1` to its component `c`. This situation is defined as follows:

```
b1.r1                              and              b2.r5
        RULE rule1                                      RULE rule2
        ON    s1                                        ON    s2
        DO    request(s2)                               DO    c->m1()
```

Based on the broker ECA-rules SAMOS ECA-rules are generated taking into account the implicit responsibilities of each broker as expressed in the above rules.
We thus have the following SAMOS statements:

```
DEFINE EVENT s1                  DEFINE EVENT s2
DEFINE RULE rule1            and              DEFINE RULE rule2
ON    s1                                      ON    s2
DO    b1->r1.request(s2)                      DO    b2->c->m1()
```

In the action part of the SAMOS ECA-rule we can have either calls to predefined broker (role) operations, or calls to methods of the passive components of the broker for which the rule is generated. Responsibilities are used when transforming broker rules to SAMOS ECA-rules in order to associate services to the roles or components that provide them. Service requests are modeled as SAMOS abstract events and are signalled by the request operations performed by brokers which send the *raise_event* message to the rule manager with the service name as parameter.

Relationships between brokers (e.g. precedence in responding to a service request, or "exactly one" agent execution semantics) are also taken into account when transforming their rules into SAMOS ECA-rules. Such transformations may introduce priorities in rule execution or mutually exclusively executing rules.

5 An Example

In this section, we present an example workflow. An example from the software process domain can be found in [22]. Consider the processing of a health insurance claim (HIC) as shown in the activity diagram in Figure 3. Once the HIC is received, a human agent creates an electronic dossier containing the diagnosis, the treatments, and costs (from the HIC), and if an insurance policy exists, a reference to the entry in the insurance company database (activity `A1`). An automatic agent controls whether there is a valid insurance policy for this HIC (`A2`). Activity `A3` controls whether the total cost is less than a certain amount (e.g. 300 Francs). In that case the HIC is directly forwarded to an automatic agent which prepares a check, prints it and notifies a clerk (`A4`). Otherwise, further controls are performed in parallel by automatic and human agents. One is whether some of the treatments are contained in a blacklist in which case their coverage will be denied (`A5`). Activity `A6` performed by an external rule-based system controls whether the (combined) treatment actually suits the diagnosis, and `A7` checks for compatibility of the diagnosis and treatment, with respect to the patient's history. If one of these controls fails, an entry is made in the customer history, a notification of the rejection is printed (`A8`) and a clerk is informed. Otherwise, a payment check is printed and a clerk is again notified. Ultimately, a law specifies that the insurance com-

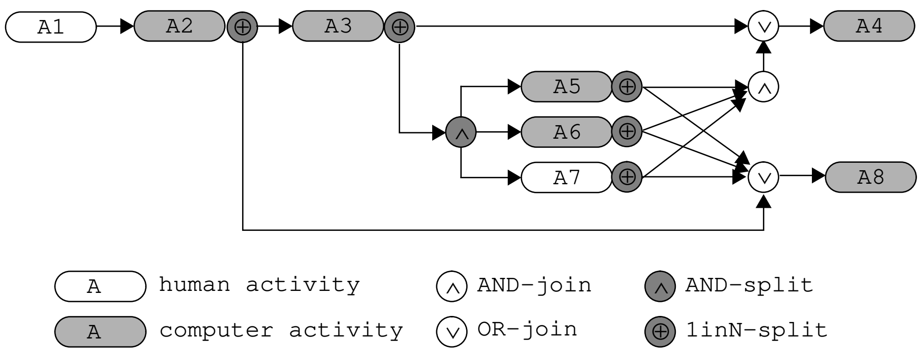

Figure 3. Sample Workflow

pany must react (either positively or negatively) at the latest after six weeks. For our example we assume that if no rejection decision has been made within six weeks from the dossier creation the claim is automatically accepted. Note that the activity diagram does not show the transitions in case the HIC is accepted due to having reached the time limit as shown in rule `accept2` below.

Parts of the broker definitions responsible for these activities are shown below in order to show a possible implementation of activity sequencing constraints (see Table 1) of the workflow described. We assume the following (incomplete) type definition for HIC dossiers:

```
TYPE HIC_DOSSIER
        amount : AMOUNT
        creation_date : DATE
        state : HIC_STATE {Rejected, Accepted, InProcess}
        insurance_policy : INSURANCE_POLICY_REF
        ...
```

A *sequence* is defined in the following two rules describing the reaction of the broker responsible for the activity `A3` (`Broker_C`) to the successful completion of `A2` by `Broker_B`. Note that depending on the outcome of `A3` either activity `A4` will take place or other checks will be performed in activities `A5`, `A6`, and `A7` (1-in-N split):

```
BROKER Policy_Checker: INTERNAL
// reaction to the request to check the claim validity
RULE  check1
ON    check_hic(hic_dossier: HIC_DOSSIER)
IF    NOT (hic_dossier.insurance_policy == NULL)
DO    reply(policy_valid, check_hic, hic_dossier)
...
BROKER Amount_Checker: INTERNAL
// reaction to a reply that the HIC refers to a valid policy
RULE  valid1
ON    policy_valid(hic_dossier: HIC_DOSSIER)
IF    hic_dossier.amount =< 300
DO    hic_dossier.state = Accepted
```

```
        reply(amount_small, check_hic, hic_dossier)
// reaction to a reply that the HIC refers to a valid policy
RULE  valid2
ON    policy_valid(hic_dossier: HIC_DOSSIER)
IF    hic_dossier.amount > 300
DO    reply(amount_large, check_hic, hic_dossier)
...
```

An *AND-join* is exemplified in the rule `accept1` of an internal broker (e.g. `Broker_D`) when the checks made in activities `A5`, `A6`, and `A7` are all positive the claim can be accepted:

```
// reaction to positive results from various checks
RULE  accept1
ON    not_blacklisted(hic_dossier_1: HIC_DOSSIER) AND compatible_treatment
      (hic_dossier_2: HIC_DOSSIER) AND compatible_history(hic_dossier_3:
      HIC_DOSSIER)
IF    hic_dossier_1 == hic_dossier_2 == hic_dossier_3
DO    hic_dossier.state = Accepted
      reply(hic_accept, check_hic,hic_dossier_1)
```

An *OR-join* is exemplified in the rule `print1` of a printer interface broker which describes the activities performed upon acceptance of the claim:

```
// reaction to acceptance of the claim
RULE  print1
ON    amount_small(hic_dossier: HIC_DOSSIER) OR
      hic_accepted(hic_dossier: HIC_DOSSIER)
DO    printer->printCheck(hic_dossier) // call method of printer component
      request(notify_clerk, print_location)
```

A *deadline* is defined with the rule `accept2` of the `Broker_D` stating that if no rejection of the HIC has been made within 6 weeks the claim will be accepted:

```
// acceptance of claim if deadline has been reached and it has not been rejected
RULE  accept2
ON    NOT(reject_hic(hic_dossier: HIC_DOSSIER, reason: REASON))
      IN [hic_dossier.creationdate + 6 weeks]
IF    hic_dossier.state == InProcess// set in A1 and since not changed
DO    hic_dossier.state = Accepted
      reply(hic_accept, check_hic, hic_dossier)
```

Service Name	Parameters	Replies
check_hic	hic_dossier: HIC_DOSSIER	policy_valid, reject_hic, amount_large, amount_small, hic_accept, not_blacklisted, compatible_treatment, compatible_history
notify_clerk	print_location: PRINTER_NAME	-

Table 2. Services and replies used in the example workflow

6 Related Work

Both WFMSs and PCDEs use some kind of “process engine” for process enactment [e.g., 6, 17]. It has been investigated for both kinds of systems how active mechanisms can be used. However, to date only ADBMSs that support primitive events have been used [e.g., 6, 7, 17], and thus control with complex constraints as described above is not possible within the ADBMS. For instance, the SPADE [1] environment is implemented on top of the ADBMS NAOS [7] and still uses external to the ADBMS process interpreters for process enactment. Adele/Tempo [3] is based on a DBMS providing an extended ER-Model. Interpreted temporal event-condition action rules are attached to software objects to define development policies and express integrity constraints. The supported event types are database operations and the conditions (defined as part of the events) are formulas over the past and present state of the system or database.

The work presented in [8] is similar to our approach in that it uses ECA-rules to control and organize long-lasting workflows. However, in the broker/services model introduced here, additional abstractions are introduced which —as we feel— serve the purpose of designing CPEs better than “pure” ADBMSs. Particularly, the broker/services model supports agent- or service-oriented views, which are not apparent if CPE-design and imlementation actually means programming an active database system.

Condition-action rules have also been used in PCDEs, e.g., in ALMA [18] or Marvel [2]. These approaches, however, are potentially less efficient (since events are not supported), and complex constraints on processes can be formulated, checked, and enforced in a less elegant way than is possible with a system supporting complex constraints attached directly to agents.

The integration of existing and possibly heterogeneous component systems is also a goal of the REACH project [4]. In contrast to our intended application domain, REACH focuses on real-time applications (where deadlines are much harder and more critical than in our types of processes). We consider the work done in REACH as complementary to ours since REACH so far has mainly considered the transaction management aspect (which is still open here).

7 Conclusion

We have described the broker model which we use for the realization of CPEs, namely control, communication, and coordination of CPE-components. In comparison to current approaches using process engines, the broker-based approach is more flexible and allows a more natural view of CPEs, since the relevant structures and behavior can be specified local to brokers. Thus, the contribution of this work is twofold:

- by using brokers, tasks related to control, communication, and coordination can be distributed among the brokers, and
- brokers can be easily implemented using an ADBMS such as SAMOS.

The broker model as presented here is currently under implementation on top of SAMOS. Two aspects of CPEs not investigated here, are subject to future work:

- a complete *programming environment* for customizing CPEs, and
- *autonomy of component systems* and *transaction management (TM)*.

First, we have described the use of brokers for the customization of CPEs. Clearly, a more abstract and declarative model (e.g., a graphical design tool) would be helpful. Most likely, we will not develop yet another language, but evaluate existing ones for our purposes. Functionalities such as planning, measurement, and process evolution shall be covered by such a programming environment as well. Additionally, this environment shall support process state representation and visualization.

Second, since we use brokers as wrappers for external components, interoperability and autonomy have to be addressed in the context of a suitable wrapper definition. In combination with TM, however, they pose a much harder problem. With respect to TM, we want to achieve the following:

- processes should be definable as long-lived transactions (e.g., comparable to DOM transactions [5]),
- TM on the level of CPEs should be able to integrate the local TM mechanisms of the component systems, wherever present.

TM in interoperable systems is still an open problem. We plan to investigate whether it can beneficially be implemented local to brokers (in case the wrapped system does not provide full-fledged TM) using our construction approach in terms of strategies and techniques [15], and the transformational approach for transaction structures [14]. We will also investigate whether the concept of strategy can be extended so that it can guide CPE-implementors during the wrapping and integration process in these cases where component systems already have a local transaction manager.

8 Acknowledgments

We gratefully acknowledge the comments and ideas contributed by Klaus Dittrich. We thank Stefan Scherrer for illuminating explanations of the health insurance business.

We also thank the Swiss Federal Office for Education and Science for funding our part in the ACTNET HCM-network (BBW Nr. 93.0313). The work of M. Kradolfer is funded by the Swiss National Fund in the context of the TRAMs project (Nr. 21-40440.94).

9 References

1. S. Bandinelli, L Fuggetta, C. Ghezzi, L. Lavazza: SPADE: An Environment for Software Process Analysis, Design and Enactment. In [9].
2. N.S. Barghouti: Supporting Cooperation in the MARVEL Process-Centered SDE. *ACM Software Engineering Notes,* 17:5, December 1992.
3. N. Belkhatir, W.L. Melo: Evolving Software Processes by Tailoring the Behavior of Software Objects. *Proc. IEEE Intl. Conf. on Software Maintenance*, Victoria, September 1994.
4. H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann: Rules in an Open System: The REACH Rule System. In [20].
5. A. Buchmann, M.T. Oezsu, M. Hornick, D. Georgakopoulos, F.A. Manola: A Transaction Model For Active Distributed Object Systems. In A.K. Elmagarmid (ed): *Database Transaction Models For Advanced Applications*. Morgan Kaufmann Publishers, 1992.

6. C. Bussler, S. Jablonski: Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems. *Proc. 4th Intl. RIDE: ADS Wokshop*, Houston, Texas, February 1994.
7. C. Collet, T. Coupaye, T. Svensen: NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. *Proc. 20th Intl. VLDB Conf.*, 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, May 1990.
9. A. Finkelstein, J. Kramer, B. Nuseibeh (eds): *Software Process Modeling and Technology.* Research Studies Press Limited, 1994.
10. S. Gatziu, A. Geppert, K.R. Dittrich: Integrating Active Concepts into an Object-Oriented Database System. *Proc. 3rd Intl. DBPL Workshop*, Nafplion, Greece, August 1991.
11. S. Gatziu, K.R. Dittrich: Events in an Active Object-Oriented Database System. In [20].
12. M.R. Genesereth, S.P. Ketchpel: Software Agents. *Communications of the ACM,* 37:7, July 1994.
13. D. Georgakopoulos, M. Hornick, A. Sheth: *A*n Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases,* 3:2, April 1995.
14. A. Geppert, K.R. Dittrich: Rule-Based Implementation of Transaction Model Specifications. In [20].
15. A. Geppert, K.R. Dittrich: Strategies and Techniques: Reusable Artifacts for the Construction of Database Management Systems. *Proc. 7th Intl. Conf. on Advanced Information Systems Engineering*, Jyväskylä, Finland, June 1995.
16. A. Geppert, S. Gatziu, K.R. Dittrich: Architecture and Implementation of an Active Object-Oriented Database Management System: the Layered Approach. TR, Computer Science Dept., University of Zurich, 1995.
17. H. Jasper: Active Databases for Active Repositories. *Proc. 10th Intl. Conf. on Data Engineering*, Houston, Texas, February 1994.
18. A. van Lamswerde: Active Software Objects in a Knowledge-Based Lifecycle Support Environment. In D. Mandrioli, B. Meyer (eds): *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.
19. F. Leymann, W. Altenhuber: Managing Business Processes as an Information Resource. *IBM Systems Journal*, 33:2, 1994.
20. W. Paton, H.W. Williams (eds): *Rules in Database Systems*. Workshops in Computing, Springer-Verlag, 1994.
21. M. Rusinkiewicz, A. Sheth: Specification and Execution of Transactional Workflows. W. Kim (ed): *Modern Database Systems*. Addison Wesley, 1995.
22. D. Tombros, A. Geppert, K.R. Dittrich: SEAMAN: Implementing Process-Centered Software Development Environments on Top of an Active Database Management System. TR, Computer Science Dept., University of Zurich, 1995.
23. *Glossary. A Workflow Management Coalition Specification*. The Workflow Management Coalition, Bruxelles, Belgium, November 1994.