A Reusable Architecture to Construct Active Database Systems

Hans Fritschi, Stella Gatziu
Technical Report 99.02
Database Technology Research Group
Institut für Informatik, Universität Zürich
Email: {fritsch|gatziu}@ifi.unizh.ch

Abstract

The construction system FRAMBOISE is intended to provide active database mechanisms by bundling them together out of existing, prefabricated components realizing specific active database tasks. The systems built by means of FRAMBOISE are built according to a common architecture. This reference architecture not only serves as a blueprint to assemble systems but establishes also a context that underlies the design of new components. This paper discusses the function of the reference architecture in the overall FRAMBOISE project and presents it formally by means of an architecture description language.

1 Introduction

The FRAMBOISE project proposes a reuse-oriented approach to construct so-called ECA-Systems (ECAS) which allow the definition and execution of active database mechanisms interoperating with specific mostly passive database management systems [FGD98]. Principally, new ECASs are provided by bundling them together out of existing, prefabricated components that realize specific active database tasks. Hence, the FRAMBOISE approach follows a general direction as it is proposed in [Vas94, Bla96, GD94] and particularly in [GD98], namely to provide individual database management services that can be used and combined in a variety of ways and in a variety of environments.

Obviously, there must be adequate components in order to build ECA Systems by putting together preexisting, reusable and compatible components in a systematic way (subsequently referred as *rebundling*). Such components are either gained by designing them from scratch or as a result from *unbundling* preexisting software systems. Unbundling is defined as the activity to decompose a category of systems into a set of reusable components as well as a set of relationships between these components [GD98]. Hereby, "decomposition" is not restricted to the meaning that concrete systems are analyzed and parts of them identified as reusable components. Instead, one would consider them beforehand on an abstract, conceptual level and determine which components would be required for such a system. Based on such a generic description of the respective components one starts to build repositories of components, and in this phase one might in fact get components out of concrete existing (active) DBMSs.

In that sense, this paper summarizes the initial unbundling activities in the FRAMBOISE project. Complex software systems like ADBMSs are usually not unbundled in one single step, but – as in any design activity – first one identifies the principal aspects to be covered and elaborates them separately later on. Accordingly, we conceive ECA Systems at the outset in a conceptual way and establish a concise, formal description of their architecture. This architectural description gives a comprehensive view of all aspects to be incorporated in FRAMBOISE and serves later on as some kind of "road map" to guide further unbundling steps. Moreover, it will be used in the rebundling process as a *reference architecture* according to which all ECA Systems are assembled.

The paper is organized as follows: In the next section we set the stage by presenting what active database functionality shall be covered potentially by ECA Systems. In Section 3 we present the architecture model used to describe the reference architecture which is subsequently elaborated in Section 4. Finally, Section 5 concludes the paper.

2 The Functionality covered by FRAMBOISE

Active database management systems principally support applications whose behavior depends on situations occurring in the databases by moving reactive behavior from the application (or a polling mechanism) into the DBMS itself. Thus, ADBMSs provide mechanisms to describe the reactive behavior (generally referred to as the *knowledge model*) as well as support for monitoring and reacting to relevant situations (addressed as *execution model*). Furthermore, ADBMSs usually provide assistance for *administration tasks* (e.g., specialized browsers or debuggers etc.).

An initial domain analysis resulted in the conclusion that the software systems built by means of FRAMBOISE rely on the Event-Condition-Action rule paradigm which is by far the most prominent approach for the knowledge model. Consequently these systems are named ECA Systems. Moreover, the following aspects are to be considered in the the future unbundling steps:

- *The definition of rules.* ECA Systems must offer constructs to define ECA-rules which specify the part of the application semantics representing the active behavior. The expressiveness and the variety of the constructs, especially those used for the specification of events, is crucial for a practical applicability of ECA Systems.

- *The management of rules.* Information about rules, events, conditions and actions is basically managed by the ECAS, which may (but are not required to) rely on features provided by the underlying DBMS to store rule definitions.

- *The detection of events.* One main feature of active DBMS is the ability to detect efficiently all events requiring a reaction. ECA Systems, however, usually perform no actual event detection of their own. On the one hand, events taking place in a DBMS (i.e., *database events* like insertion of a tuple or changing the attribute value of an object) should be monitored and detected within the respective DBMS. On the other hand *external events* are typically signalled explicitly (i.e., *raised*) by applications outside the ECASs[1] However, *complex events* are detected by adequate

[1]In practice this assumption may be weakened, because it is conceivable to provide in FRAMBOISE "ready-to-use" components which perform the detection of the so-called *time events* whose occurrence originate also from outside an ECAS (i.e., the system clock).

complex event detectors which are an integral part of an ECAS.

- *The execution of rules.* Upon an event signalisation the ECAS reacts with the execution of the rules defined for the respective events. How this execution takes place is determined by several predefined or user-defined guidelines which form the execution model of the respective ECAS. It must be possible to provide ECAS that allow conditions to be evaluated and actions to be executed either immediately upon event detection or at a later stage upon signalisation of a *rule assertion point.* Moreover, it must be possible to allow *detached rule execution* i.e., condition evaluation as well as action execution may be performed in separate transactions.

Note, that this are only the relevant function points to identify the chief aspects of FRAMBOISE. Having established the reference architecture, these aspects will subsequently be unbundled themselves, based on a detailed analysis dedicated to the respective aspect.

3 The Software Architecture Model

A *software architecture* is a description of a system structure and decomposes the system into well-defined components as subsystems and relationships between them. Software architectures are in turn formulated by means of a *software architecture model.* The most important ingredients of such an architectural model define the notion of component as basic units, relationship types between these components and an architectural style (e.g., layered architecture).

In order to describe the reference architecture of the ECA Systems we adopt an architectural model from [AG94, AG97], consisting of *components*, *connectors* between components and *policies* restricting the ways how components and connections are arranged and how they interact. Components are the primary computational and storage entities of a system e.g., tools, databases, filters, servers etc. Connectors capture the policies governing the cooperative behavior of the components by regulating the interaction among a collection of components. These architecture elements typically stand for rather complex software artifacts, hence components as well as connectors may be decomposed further into (sub) components which are in turn related via other connectors.

Architectures consisting of components and connectors can be depicted by means of ER-style diagrams, denoting components as rectangular boxes and connectors as diamond-shaped relationships (cf. Fig 2). Such graphical representations accompanied with colloquial descriptions provide practical documentation, however, their informality limits their usefulness. It is rather difficult to analyze an architecture for consistency or to determine non-trivial properties of it, because it is typically not feasible to state precisely what is meant by such informal architecture descriptions.

The Architecture Specification Language WRIGHT We apply the architecture description language WRIGHT [AG94, AG97, All97] to specify the reference architecture of FRAMBOISE. WRIGHT supports reasoning and formal manipulation but is nevertheless a vehicle of expression that matches the intuitions and practices of a software designer. The language is built around the basic architectural abstractions of *components*, *connectors* and *configurations*, providing explicit structural notations for each of these elements. The general notion of component is formalized as localized independent computation whereas a connector is defined as a pattern of interaction among components. The description of a component consists of the *computation* part and of an interface part formed by a number of *ports*. Each port defines a set of interactions in which the respective component may participate. Connector descriptions consist of a set of *roles* (constituting the connectors interface) and the so-called *glue*. Each role defines the behavior of one component in the interaction, whereas the glue defines how the roles will interact with each other.

In order to describe complete system architectures, the components and connectors of a WRIGHT description are combined into a *configuration*. A configuration is a collection of component instances combined via connectors as shown in Fig. 1. The behaviour

Configuration ClientServer
  Component DBServer =
    Port Session [*Session protocol*]
    Computation [*Server specification*]
  Component AppClient =
    Port DBRequest [*request protocol*]
    Computation [*Application specification*]
  Connector C-S-connector =
    Role Client [*client protocol*]
    Role Server [*server protocol*]
    Glue [*glue protocol*]
 Instances
  DBMS: DBServer
  Application: Client
  cs: C-S-connector
 Attachements
  DBMS.Session **as** cs.Server
  Application.DBRequest **as** cs.Client
End ClientServer.

Figure 1: A Simple Client-Server Configuration

and coordination of components and connectors is formally specified in WRIGHT using a notation based on the process algebra CSP[Hoa85].

The Process Algebra CSP CSP ("Communicating Sequential Processes") was originally devised by C.A.R. Hoare [Hoa85] as a notation and theory to describe systems as a number of elements (processes) which operate independently[2] and communicate with each other over well-defined channels. Besides being a notation for describing *concurrent systems*, CSP is also a collection of mathematical models and reasoning methods to understand this notation.

The basic notion of CSP is the so-called *process* which stands for the behaviour pattern of any real world object that acts and interacts with other objects according to a specific

[2]The restriction that processes must be sequential was removed between 1978 and 1985, but the name CSP was already established.

manner. The behavior pattern of such an object is described by means of *events* which are abstractions of actions performed in the real world. In order to describe processes whose behaviour is influenced by interaction with their environment (represented by other processes), CSP includes various primitive operators for parallel composition, nondeterministic as well as deterministic choice. This makes for an elegant notation in which the problems of concurrency, nondeterminism and abstraction can be addressed separately. The language also provides constructs for modeling deadlock and recursion.

Processes are denoted in *upper-case* words and events in *lower-case* words. The set of names of events which are considered as relevant for a specific process is called its *alphabet* and denoted with α (e.g., the alphabet of a process P is addressed as αP). The special process $STOP_{\alpha P}$ is a process that engages in no events of the alphabet αP. It describes a broken process i.e., a useless process that results probably from a deadlock or other design error, rather than a deliberate choice of the designer. The *success event* $\surd$, however, is a *special event* that stands for the successful termination of a process. Note that an event name actually denotes an event classe which is distinct from the event occurrences, since a specific event may occur any number of times. The actual occurrence of each event in the life of a process is regarded as an instantaneous or atomic action without duration[3]. An important property of events is that they can carry data. If a process supplies data, this is considered as output and written with an exclamation point e.g., $write!x$. If a process receives data, this is input and written with a question mark: $read?x$.

The *prefix* operator $\rightarrow$ allows to enhance process descriptions by communication events. For instance, the process $x \rightarrow P$ is initially prepared to engage first the event x and afterwards behaves like P. The notation $P \,\Box\, Q$ represents an *external choice* between the two processes P and Q. If the environment is prepared to interact with P but not Q, then the choice is resolved in favour of P and vice versa. $P \sqcap Q$ is an internal choice between P and Q; the outcome of this choice is nondeterministic.

WRIGHT extends CSP in some minor syntactic ways. For instance, it distinguishes between *initiating* and *observing* an event. An event that is initiated by a process is written with an overbar (e.g., the database client in the example above indicates with $\overline{query!x}$ that it initiates the query wheras it waits for the result of the query). Morevoer, WRIGHT introduces the special event $\S = \surd \rightarrow STOP$, because the CSP special event $\surd$ typically only occurs in a process that halts immediately after indicating termination. For example an application that emits exclusively queries against a database server can be described by the following process:

$$\textbf{Role}\ \text{QueryClient} = \overline{query!x \rightarrow result?y \rightarrow QueryClient \sqcap \S}$$

The CSP-based descriptions of connectors are sufficient to determine whether the glue constrains the roles enough to ensure properties like the local absence of deadlocks. moreover, the descriptions can be used to determine if a configuration is constructed correctly i.e., whether the interfaces of a component are appropriate for a use in a particular role. The global behavior of a configuration is constructed from the processes introduced by the components and connector definitions. The events are communicated via the connectors to which they are attached, causing the glue of a connector to enforce its protocol on the communication.

[3]Extended or time-consuming operations are modelled by a pair of events, the first denoting the start of an action and the second denoting its finish.

4 The Reference Architecture

The FRAMBOISE construction system is designed up front for software reuse. However, software reuse is not restricted to code fragments, because writing code is usually down in the 10 - 20 percent range of total development costs [JGJ97]. In fact, virtually any result obtained in a previously constructed system might be reused in later constructions (e.g., designs, specifications, test cases etc.). Hence software architectures are also related to reuse, since they actually form a particular reuse technique [Kru92]. Software architectures represent significant design decisions that are reused as a whole.

Assembling ECA systems according a common architecture supports the application of generative techniques that require principally that the final products are aligned according a common blueprint. Besides, the construction process is also more cost-efficient where generative approaches are not applicable, because the intrinsic similarity of the various ECASs promotes the knowledge transfer between developers as well as it is easier for the implementors of ECAS to reuse their experience. Hence, FRAMBOISE requires a generic yet adaptable architecture according to which specific ECASs are built.

Domain analysis revealed the principal feasibility of such common architectures for unbundled active database mechanisms (cf. [GKvBF98]), since the high-level system architecture of many research prototypes incorporate similar components even though their knowledge and rule execution models are very different [PD97, WC96]. Consequently, [GKvBF98] proposes various generic architectural models that can be used as a starter-kit to initiate the unbundling of active database mechanisms for a specific purpose. The unbundling process of FRAMBOISE bases on such an architectural proposal which is presented informally in the next section. Afterwards, in section 4.2 we give the formal description expressed in WRIGHT.

4.1 Informal Description

In order to achieve a reference architecture that is small but applicable to a broad spectrum of ECAS, we applied an architecture model proposed in [GKvBF98] (cf Fig. 2) that separates the active functionality from the DBMS as an *activity service* (shaded box in Fig. 2).

The interaction of the activity service with its environment is on the one hand regulated by the *external event detection*, *external action execution* and the *rule base modification* connector. The latter allows clients to modify a rule base (e.g., to add rules). The external event detection connector gives external clients (e.g., the system clock or an application program) the opportunity to raise events of their own (*abstract* events). The external action execution connector enforces the execution of actions taking place at some external client.

On the other hand, the *database event detection*, *condition evaluation* and the *database action execution connector* regulate the interaction between activity service and the DBMS. Establishing this interaction by means of well-defined connectors enables the design of a "core" active functionality – subsequently addressed as *generic ECAS* – which needs to be developed only once and can be applied in various environments and adapted for specific application profiles as well. Thus, generic ECASs – which are to be distinguished according their specific knowledge and rule execution model – are most promising subjects of reuse. Basically, a generic ECAS consists of two major components i.e., of an *event* and

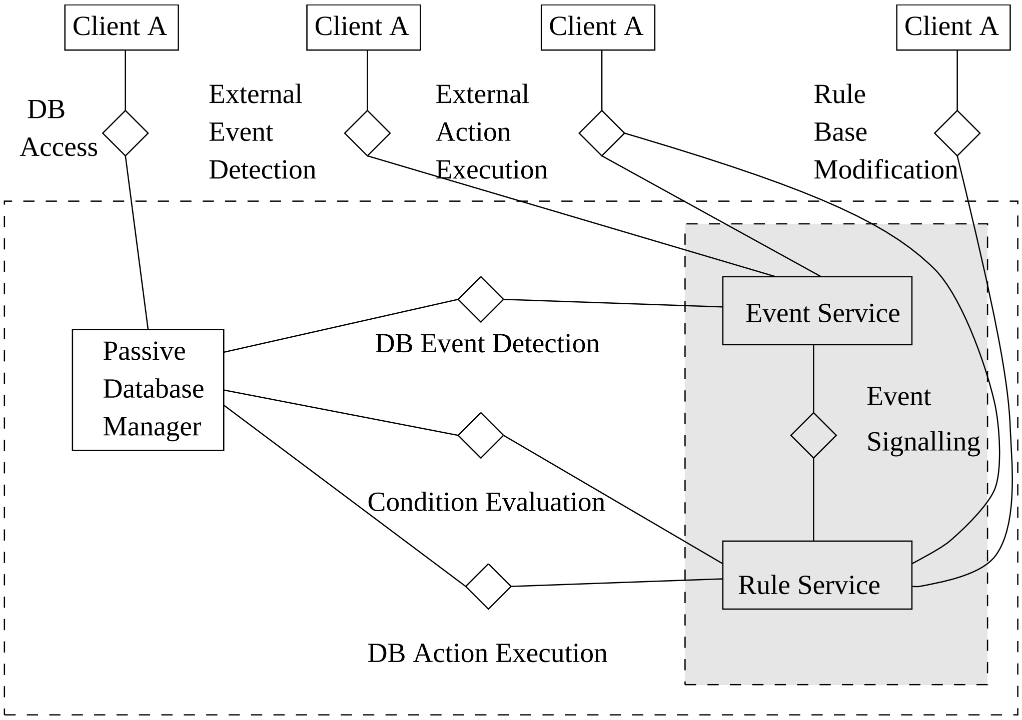

Figure 2: The Reference Architecture of ECA Systems.

a *rule service*. The event service records events, maintains a (persistent) event history (which consists of all event occurrences which have not yet been consumed for a rule execution or composite event detection) and detects composite events. The rule service is responsible for the maintenance of the rule base and implements the rule execution cycle. Both components i.e., the event and the rule service, are related by means of an *event signalling connector*.

The database event detection connector regulates the interaction between database related event detection mechanisms (e.g., proprietary event notification mechanisms that reside in the respective DBMS) and the generic ECAS. It determines what type of events the DBMS produces and to what events the event service subscribes to. Typically, such a connector performs the mapping of the DBMS-specific event parameters to a format used by the generic ECAS. According to the event semantics and rule execution guidelines, the connector may additionally enforce further activities in order to synchronize the DBMS and the rule processing task. For instance, it might block the triggering transaction until all triggered rules are processed.

In order to evaluate conditions and to execute database actions, the generic ECAS cooperates with the database manager via the condition evaluation and the database action execution connectors. These connectors initiate the DBMS to perform specific queries or DML statements and inform the activity service subsequently about the outcome of the specific operations. Note that these activities might be executed as part of the triggering transactions, which requires further coordination by the connectors.

4.2 The Formal Description

The informal description of the reference architecture given in section 4.1, brushes a lot of important information under the rug and does not provide enough aspects to begin a detailed design. For instance, this kind of abstraction is not concise enough to resolve subsequently how an ECA System and a specific DBMS are synchronized in order to process rules triggered by database events. A WRIGHT-based description of this "naive" specification, however, enables a designer to reason about the complicated interactions as they emerge when ECA Systems are confronted with specific database, operating system or application issues. Solutions can then be expressed on an architectural level as systems of refined WRIGHT components and connectors that replace or enhance generic elements in the basic reference architecture.

In order to prevent a designer to fail the wood for the trees, the basic WRIGHT description of the reference architecture must not be cluttered with overly detailed and lengthy specifications that are difficult to grasp. Consequently some architectural aspects should be treated to a lesser extent in order to achieve a compact presentation. Providing up front complete specifications of the parts belonging to the generic ECASs (e.g., rule service) were feasible but premature. First, it is necessary to analyze how ECA systems may cooperate with concrete DBMS and system environments to conclude in which aspects of the knowledge and execution model various ECASs actually may differ. Only then it is sensible to specify the subsystems of the generic ECAS comprehensively, because the insights of the previous analysis are necessary to design event and rule service components that must be configurable to provide the behaviour of different knowledge and execution models.

Therefore, the following basic specification is outlined to underpin a future reasoning about the interaction of ECA Systems with their environment (e.g., specific DBMSs, applications, operating system services etc.). We focus on the specification of the interface parts (i.e., ports of components and roles of connectors) of the architectural elements. They are specified completely and – as far as they are directly related to generic ECASs – should not undergo major revisions. Internals of the event and rule service like the maintenance an event history or the existence of a rulebase are not specified as components of their own but are merely addressed as specific events or processes. Moreover, the treatment of incorrect commands (e.g., an illegal rule definition) is not modeled at this level of abstraction. For the time being it is sufficient to think in terms of a rulebase administrator trying out successive communications until finding the correct ones. Accordingly it is safe to assume that signalled event occurrences are always matched by corresponding definitions in the rulebase. Furthermore, we summarize inserting, deleting or updating elements in the data- or rulebase in CSP as modifications.

In the following paragraphs the WRIGHT specification of the various architectural elements are presented in detail.

4.2.1 The Rule Service

The rule service is the core facility of any ECA System that maintains the rule base and implements the rule execution cycle. It cooperates with the event service, the (passive) database manager, with programs that are launched as a consequence of rule execution and finally with utilities to maintain the rulebase (e.g., rulebase editors, rule compilers etc.). Thus, it is sensible to devise the rule service also as the central coordinator of an ECAS. The rule service is structured as follows:

Component RuleServiceService =
    Port Administration = ...
    Port Event = ...
    Port ExternalAction = ...
    Port Condition = ...
    Port DBAction = ...
    Computation = ...

In the following paragraphs, we examine the various facets of the rule service in detail, starting with the ports and ending with the computation.

The Administration Port This port enables a client to start and to halt ECAS and to communicate rulebase modification commands.

$$
\begin{aligned}
&\textbf{Port}\ \text{Administration} = \mathit{start}?\mathit{session} \rightarrow \mathit{Run}\\
&\quad\textbf{where}\quad \mathit{Run} = \mathit{modifyEvent}?\mathit{evDef} \rightarrow \overline{\mathit{return}} \rightarrow \mathit{Run}\\
&\qquad\qquad\quad \Box\ \mathit{modifyCondition}?\mathit{conDef} \rightarrow \overline{\mathit{return}} \rightarrow \mathit{Run}\\
&\qquad\qquad\quad \Box\ \mathit{modifyAction}?\mathit{acDef} \rightarrow \overline{\mathit{return}} \rightarrow \mathit{Run}\\
&\qquad\qquad\quad \Box\ \mathit{close} \rightarrow \S
\end{aligned}
$$

The Event Port The rule service communicates with the event sources through this port. Upon connecting to the event sources, the rule service is able to receive event signals (`signal`) as well as to order the event source to begin respectively to stop supervising specific event occurrences (represented by means of the CSP event `monitor`).

$$
\begin{aligned}
&\textbf{Port}\ \text{Event} = \overline{\mathit{connect}!\mathit{session}} \rightarrow \mathit{Listen}\\
&\quad\textbf{where}\ \mathit{Listen} = ((\mathit{signalEvent}?\mathit{instances}\ \Box\ \overline{\mathit{processed}!\mathit{occ}}\ \Box\ \mathit{ruleassert}\\
&\qquad\qquad\quad \Box\ \overline{\mathit{monitor}!\mathit{evDef}}) \rightarrow \mathit{Listen})\\
&\qquad\qquad\quad \Box\ \overline{\mathit{close}} \rightarrow \S
\end{aligned}
$$

The CSP event *ruleassert* that represents signalling so-called *rule assertion point* that forces the execution of all pending rules that are currently scheduled to be executed deferred.

The External Action Port External actions are arbitrary executable programs which are invoked as a consequence of rule execution. Besides directing external action execution at rule execution time, the rule service enforces the availability of the executable program at rule definition time via this interface.

$$
\begin{aligned}
&\textbf{Port}\ \text{ExternalAction} = \overline{\mathit{connect}!\mathit{session}} \rightarrow \mathit{Operate}\\
&\quad\textbf{where}\quad \mathit{Operate} = \overline{\mathit{exec}!(\mathit{evInst}, \mathit{acDef})} \rightarrow \mathit{return} \rightarrow \mathit{Operate}\\
&\qquad\qquad\quad \Box\ \overline{\mathit{register}!\mathit{acDef}} \rightarrow \mathit{return} \rightarrow \mathit{Operate}\\
&\qquad\qquad\quad \Box\ \overline{\mathit{close}} \rightarrow \S
\end{aligned}
$$

The Condition Port Conditions are evaluated outside the rule service. Thus, the condition port is an interface that permits the rule service to order an external component to evaluate a condition. The executable code representing the condition is maintained by this external component. Thus, the rule service orders this component through the condition port to register a condition in order to translate it into an executable format. Since the condition is usually a query performed by a database server and the knowledge model of an ECAS may incorporate decoupled rule execution, the rule service is enabled to start and end transactions via the condition port.

$$
\begin{aligned}
&\textbf{Port}\ \text{Condition} = \overline{establish!session} \rightarrow Operate\\
&\quad \textbf{where}\ Operate =\\
&\qquad\qquad (EvalQuery \sqcap \overline{bot} \rightarrow EvalQuery \sqcap \overline{bot} \rightarrow EvalQuery \rightarrow \overline{eot});\ Operate\\
&\qquad\qquad \Box\ (\overline{register!conDef} \rightarrow return \rightarrow Operate)\\
&\qquad\qquad \Box\ (\overline{close} \rightarrow \S)\\
&\qquad \textbf{where}\ EvalQuery = \overline{evaluate!(evInst, conDef)} \rightarrow return?result
\end{aligned}
$$

Note that we express the decision whether a condition is evaluated decoupled from the triggering transaction as an *internal* choice, because the rule service chooses the respective mode autonomously on account of the respective rules definition.

The Database Action Port The database action port is similar to the condition port with the difference that there are no queries to be evaluated but DML statements are performed.

$$
\begin{aligned}
&\textbf{Port}\ \text{DBAction} = \overline{establish!session} \rightarrow Operate\\
&\quad \textbf{where}\ Operate =\\
&\qquad (exec!(evInst, acDef) \sqcap \overline{bot} \rightarrow exec!(evInst, acDef)\\
&\qquad \sqcap\ \overline{bot} \rightarrow exec!(evInst, acDef) \rightarrow \overline{eot});\ Operate\\
&\qquad \Box\ (\overline{register!acDef} \rightarrow return \rightarrow Operate)\\
&\qquad \Box\ (\overline{close} \rightarrow \S)
\end{aligned}
$$

The Computation The ports of the rule service indicate already some aspects of this component, namely the properties the rule service must have if is viewed through the lens of a particular port. Thus, a port becomes a *partial specification* and the *computation* provides accordingly a complete description of what is done by integrating the views of the different ports. The following paragraphs provides a description of a rather simple but nevertheless feasible rule service that coordinates first the initialization of the ECA System and handles subsequently the various incoming commands and event signals. Note, that the events which are communicated to and from specific ports are prefixed by the name of the respective port.

$$
\begin{aligned}
&\textbf{Computation} = (Administration.start?session \rightarrow (Initialize;\ MainActivity)\\
&\quad \textbf{where}\ \text{Initialize} = \overline{Event.connect!session} \rightarrow \overline{Condition.establish!session}\\
&\qquad \rightarrow \overline{DBAction.establish!session} \rightarrow \overline{ExternalAction.connect!session})\\
&\quad \textbf{where}\ \text{MainActivity} = \ldots
\end{aligned}
$$

In the main activity process, the rule service waits for input which either may come from the `Event` (i.e., the signalling of an event occurrence) or from the `Administration` port as (i.e., rulebase modification commands or the order to halt the ECAS). Upon event signalling, the rule service executes the rules that are defined for the respective events. As explained above, the rule schema is not specified directly. Therefore, all decisions depending the contents of the rule schema are subsequently regarded as non-deterministic choices (represented by the *internal choice* operator $\sqcap$). Correspondingly, the decision whether the rule service has to engage in the rule execution process or has to recur immediately to the main activity process is an internal choice of this component.

where MainActivity =
$\quad Event.signalEvent?instances \rightarrow f_{rules}(instances) \rightarrow$
$\quad (RuleExecution \sqcap \overline{Event.return!occ} \rightarrow MainActivity)$
$\qquad$ **where** RuleExecution = ...
$\quad \Box\ Event.ruleassert \rightarrow (DeferredRuleExec \sqcap MainActivity)$
$\qquad$ **where** DeferredRuleExec = ...
$\quad \Box\ Admin.modifyEvent?ev \rightarrow ModifyEvent$
$\qquad$ **where** ModifyEvent = ...
$\quad \Box\ Admin.modifyCondition?cond \rightarrow ModifyCondition$
$\qquad$ **where** ModifyCondition = ...
$\quad \Box\ Admin.modifyAction?act \rightarrow ModifyAction$
$\qquad$ **where** ModifyAction = ...
$\quad \Box\ Close$
$\qquad$ **where** Close = $Administration.close \rightarrow$
$\qquad\quad \overline{Event.close} \rightarrow \overline{Condition.close} \rightarrow \overline{DBAction.close} \rightarrow$
$\qquad\quad \overline{ExternalAction.close} \rightarrow \overline{Administration.return} \rightarrow \S)$

In the following section we present a specification of a limited rule execution process. We assume the process to execute one triggered rule after the other until the set of triggered rules is empty. The knowledge model of this rule service allows for immediate, deferred and decoupled rule execution.

where RuleExecution = $(\overline{defer!(evInst, ruleDef)} \rightarrow \overline{Event.return}) \sqcap$
$\quad (EvalCond \sqcap (\overline{bot} \rightarrow EvalCond) \sqcap \overline{bot} \rightarrow EvalCond \rightarrow \overline{eot})$
$\qquad$ **where** EvalCond = $\overline{Condition.evaluate!(evInst, conDef)} \rightarrow$
$\qquad\quad return?result$
$\quad$ **if** $result \neq \emptyset$ **then**
$\qquad (\overline{defer!(evInst, acDef)} \sqcap$
$\qquad ExecAction)$
$\quad$ **else** $\overline{Event.return} \rightarrow RuleExecution \sqcap empty$
$\quad$ **where** ExecAction = $\overline{ExternalAction.exec!(evInst, acDef)} \rightarrow return$
$\qquad \sqcap ((\overline{DBAction.exec!(evInst, acDef)} \rightarrow return)$
$\qquad \sqcap (\overline{DBAction.exec!(evInst, acDef)} \rightarrow return \rightarrow \overline{eot})$
$\qquad \sqcap (\overline{bot} \rightarrow \overline{DBAction.exec!(evInst, acDef)} \rightarrow return \rightarrow \overline{eot}))$

Deferred rule execution is identical with the exception that conditions are not deferred anymore for later evaluation:

where DeferredRuleExec =
$\quad (EvalCond \sqcap (\overline{bot} \rightarrow EvalCond) \sqcap \overline{bot} \rightarrow EvalCond \rightarrow \overline{eot})$

where EvalCond = ...
if $\mathit{result} \neq \emptyset$ **then**
$\mathit{ExecAction})$
$\rightarrow \mathit{DeferredRuleExec} \sqcap \mathit{empty}$
where ExecAction = ...

Rulebase modification commands from the `Administration` port are chiefly processed by the rule base facilities which are not further considered at this level of detail. However, since the modification of the rulebase may propagate to various other components (e.g., the event service has to be ordered to monitor a new event) we must address some of these aspects also.

where ModifyEvent = $\mathit{Admin.modifyEvent?ev} \rightarrow \overline{\mathit{store!ev?evDef}} \rightarrow$
$\overline{\mathit{Event.monitor!evDef}} \rightarrow \overline{\mathit{return}}$

where ModifyCondition = $\mathit{Admin.modifyCondition?cond} \rightarrow$
$\overline{\mathit{store!cond?conDef}} \rightarrow \overline{\mathit{Condition.register!conDef}} \rightarrow \overline{\mathit{return}}$

where ModifyAction = $\mathit{Admin.modifyAction?act} \rightarrow \overline{\mathit{store!act?acDef}} \rightarrow$
$((\overline{\mathit{ExternalAction.register!acDef}} \sqcap \overline{\mathit{DBAction.register!acDef}} \rightarrow \overline{\mathit{return}})$

From the point of view of the interaction within the overall system, the modification of rule definitions can be subsumed under the cases illustrated so far. Therefore we leave it apart.

4.2.2 The Event Service

The event service records event notifications, maintains a persistent event history (consisting of all event occurrences which have not yet been consumed for a rule execution or composite event detection) and detects composite events. Moreover, it service transmits modifications of event definitions to the respective event sources (DBMS and system environment). The event service is structured as follows:

Component EventService =
Port Database = ...
Port External = ...
Port Rules = ...
Computation = ...

In the following paragraphs, the elements of the event service are presented in detail.

The Interaction with the Event Sources There are two event sources outside the event service, namely the DBMS and the system environment. In both cases the event service is either informed by them about an event occurrence or the service orders them to monitor the occurrence of specific events. Since the interaction proceeds in both cases according the same schema this interaction can be described by means of a so-called *interface type*. This is an independent process specification which can be applied to define subsequently ports and roles of concrete components or connectors by simply "typing" them instead of writing down each time the specification of a CSP process.

Interface Type EventSourceListener $= \overline{\mathit{establish!session}} \rightarrow \mathit{Listen}$
  where Listen $= \mathit{notify?occ} \rightarrow \mathit{Listen} \;\Box\; \overline{\mathit{processed!occ}} \rightarrow \mathit{Listen}$
    $\Box\; \overline{\mathit{monitor!evDef}} \rightarrow \mathit{return} \rightarrow \mathit{Listen}$
    $\Box\; \overline{\mathit{close}} \rightarrow \S$

An interface of the type *EventSourceListener* receives event notification or informs the event source that a specific event occurrence has been processed (i.e., all associated rules are executed). Moreover, it orders the event source to notify occurrences of a specific event. The ports `Database` and `External` are specified as *EventSourceListener*:

Component EventService =
  Port Database = EventSourceListener
  Port External = EventSourceListener
...

Besides simplifying the writing of concrete definitions of components interface types enable also the definition of complete architectural styles by defining ports and roles in a reusable way.

The Rules Port The rule port is effectively the counterpart of the database and the external port. Thus, it forwards event notifications and receives the order to register specific events for subsequent monitoring.

Port Rules $= \mathit{start?session} \rightarrow \mathit{Operate}$
  where $\mathit{Operate} = \overline{\mathit{signal!instances}} \rightarrow \mathit{return} \rightarrow \mathit{Operate}$
    $\Box\; \mathit{processed?occ} \rightarrow \mathit{Operate}$
    $\Box\; \mathit{register!evDef} \rightarrow \overline{\mathit{return}} \rightarrow \mathit{Operate}$
    $\Box\; \mathit{close} \rightarrow \S$

Since a primitive event can accomplish a complex event occurrence, the rules port can signal multiple event occurrences (the primitive one and those detected complex events).

The Computation The event service is principally a mediator between the event sources and the rule service, Hence it forwards chiefly commands from the event source listeners to the rules port and vice versa.

$$
\begin{array}{l}
\textbf{Computation} = (\mathit{Rules.start?session} \rightarrow \overline{\mathit{Database.establish!session}} \rightarrow \\
\qquad\qquad \overline{\mathit{External.establish!session}});\ \mathit{Listen} \\
\quad \textbf{where}\ \text{Listen} = (\mathit{Database.notify?occ} \ \Box\ \mathit{External.notify?occ}) \rightarrow \\
\qquad\qquad \rightarrow \overline{\mathit{Rules.signal!}(\mathit{occ} \cup \mathit{det}_{\mathit{complex}}(\mathit{occ}))} \rightarrow \mathit{Listen} \\
\qquad \Box\ \mathit{Rules.processed?occ} \rightarrow \\
\qquad\qquad (\overline{\mathit{Database.processed!occ}} \sqcap \overline{\mathit{External.processed!occ}}) \rightarrow \mathit{Listen} \\
\qquad \Box\ (\mathit{Rules.modifyEvent?evDef} \rightarrow (\overline{\mathit{Database.monitor!evDef}} \sqcap \\
\qquad\qquad \overline{\mathit{External.monitor!evDef}}) \rightarrow \mathit{return} \rightarrow \mathit{Listen}) \\
\qquad \Box\ \mathit{Rules.close} \rightarrow \overline{\mathit{Database.close}} \rightarrow \overline{\mathit{External.close}} \rightarrow \S
\end{array}
$$

The function $det_{complex}(e)$ stands for a complex event detector i.e., it returns all instances of complex events that occur directly or indirectly (due to complex events) on account of event e.

4.2.3 The Event Signalling Connector

The event signalling connector regulates the interaction between the event and the rule service. It appears in two roles as detector and as (rule) processor. The former role is attached at the *Event* port of the rule service and the latter at the *Rules* port of the event service. The following specification establishes the event signalling connector as a simple bidirectional pipe.

$$
\begin{array}{l}
\textbf{Connector}\ \text{EventSignalling} = \\
\quad \textbf{Role}\ \text{Detection} = \mathit{connect?session} \rightarrow \mathit{Listen} \\
\qquad \textbf{where}\ \text{Listen} = \overline{\mathit{signalEvent!instances}} \rightarrow \mathit{Listen} \\
\qquad\qquad \Box\ \mathit{processed?occ} \rightarrow \mathit{Listen} \\
\qquad\qquad \Box\ \mathit{register?evDef} \rightarrow \mathit{Listen} \\
\qquad\qquad \Box\ \mathit{close} \rightarrow \S \\
\quad \textbf{Role}\ \text{Processing} = \overline{\mathit{start!session}} \rightarrow \mathit{Operate} \\
\qquad \textbf{where}\quad \text{Operate} = \mathit{signal?instances} \rightarrow \overline{\mathit{return}} \rightarrow \mathit{Operate} \\
\qquad\qquad\quad \Box\ \overline{\mathit{register!evDef}} \rightarrow \mathit{return} \\
\qquad\qquad\quad \Box\ \overline{\mathit{close}} \rightarrow \S \\
\quad \textbf{Glue} = \mathit{Processing.connect?session} \rightarrow \overline{\mathit{Detection.start!session}} \rightarrow \\
\qquad\qquad \mathit{Wait4Event} \\
\qquad \textbf{where}\ \text{Wait4Event} = \\
\qquad\qquad \mathit{Detection.signal?instances} \rightarrow \overline{\mathit{Processing.signal!instances}} \rightarrow \mathit{Wait4Event} \\
\qquad\qquad \Box\ \mathit{Processing.processed?occ} \rightarrow \overline{\mathit{Detection.processed!occ}} \rightarrow \mathit{Wait4Event} \\
\qquad\qquad \Box\ (\mathit{Processing.register?evDef} \rightarrow \overline{\mathit{Detection.register?evDef}} \rightarrow \mathit{Wait4Event} \\
\qquad\qquad \Box\ (\mathit{Processing.close} \rightarrow \overline{\mathit{Detection.close}} \rightarrow \S)
\end{array}
$$

It is conceivable that the event signalling collectors specification might be extended in future to handle complex interprocess communications to synchronize the event and the rule service.

4.2.4 The Interaction of ECA Systems with the Environment

The rule and the event service are devised as system independent components, deployable identically for different ECASs that interoperate with different DBMSs and applications.

Therefore, both services are shielded from peculiar environment issues by the event detection, condition evaluation, action execution and the rulebase modification connectors which are developed to engage with specific DBMSs or operating systems. It is hence sensible to design families of system-specific connectors (e.g., a suite of database connectors for ORACLE that cooperates with triggers and PL/SQL, another for ObjectStore that copes with its event notification mechanism etc.). The presentation of concrete elements that include all intricate system-specific aspects, however, goes beyond the scope of this paper. Instead, we provide subsequently a number of simplified examples, confining ourselves to connectors. For the time being the expected behaviour of the passive database manager and the applications (i.e., client A in Fig. 2) is sufficiently described by corresponding roles being part of the connectors.

The External Event Detection Connector The following example allows an unlimited number of external clients to connect at the same time to the ECA System. This can be expressed in WRIGHT by *parameterizing* a type description. This connector is parameterized by the number of external clients (i.e., event sources) that raise external events. Note that there is not a single role *Source*, but there are as many individual roles of this type as the parameter *numSources* indicates.

Connector ExternalEventDetection(numSources: 1 ...) =
    Role $\mathrm{Source}_{1\ldots numSources} = \overline{connect} \rightarrow RaiseEvents$
        where $\mathrm{RaiseEvents} = \overline{raise!event} \rightarrow return \rightarrow RaiseEvents$
            $\Box\ disconnect$
    Role $\mathrm{EventService} = \overline{establish!session} \rightarrow Listen$
        where $\mathrm{Listen} = notify?occ\ \Box\ \overline{processed!occ}$
        $\Box\ \overline{monitor!evDef} \rightarrow return$
        $\Box\ \overline{close} \rightarrow \S$
    Glue $= EventService.establish?session \rightarrow Operate$
        where $\mathrm{Operate} = EventService.monitor?evDef \rightarrow \overline{return} \rightarrow Operate$
            $\Box\ EventService.close \rightarrow \forall\, x \in 1 \ldots numSources \bullet \overline{Source_x.disconnect} \rightarrow \S$
            $\Box\ x : 1 \ldots numSources \bullet Source_x.connect \rightarrow Push_x$
        where $\mathrm{Push_x} = (Source_x.raise?event \rightarrow \overline{EventService.notify.f_{occ}(event)} \rightarrow$
            $processed.f_{occ}(event) \rightarrow \overline{return} \rightarrow Push)$

The function $f_{occ}(event)$ returns the event instance representing the event raised by an event source.

The External Action Execution Connector The following specification describes a simple action execution connector conceived to execute external actions as independent programs that are invoked by means of the operating system shell.

Connector ExternalActionExexution =
    Role $=$ ECAS $\overline{connect!session} \rightarrow Run$
        where $\mathrm{Run} = \overline{execute!(evInst, acDef)} \rightarrow return \rightarrow Run$
            $\Box\ \overline{register!acDef} \rightarrow return \rightarrow Run$
            $\Box\ \overline{close} \rightarrow \S$
    Role $\mathrm{Shell} = \overline{system!commandString} \rightarrow return \rightarrow ECAS$
    Glue $= ECAS.connect?session \rightarrow Operate$

$$
\begin{array}{l}
\textbf{where } \text{Operate} = \mathit{ECAS.execute?!(evInst, acDef)} \rightarrow \\
\quad \overline{\mathit{Shell.system.f_{command}(evInst, acDef)}} \rightarrow \mathit{Shell.return} \rightarrow \mathit{Operate} \\
\qquad \Box\ \mathit{ECAS.close} \rightarrow \S
\end{array}
$$

The CSP event *system* stands for the UNIX system call "`system(...)`" that issues a string as a shell command. The function $f_{command}(evInst, acDef)$ returns the string that contains the shell command representing the respective action.

Interaction with the DBMS The interaction with the passive database manager is regulated on the one hand by the database event detection and on the other hand by the condition evaluation and the database action execution connector. Since the DBMS is responsible for the actual condition evaluation as well as the database action execution, the latter two connectors have mostly identical requirements. They need to establish a session with the database manager, to start and end transactions and to query or modify items (relations, objects etc.) within the database. Concurrent client accesses are thereby synchronized by the database manager. The following specification of a simple condition evaluation connector corresponds in the role *ECAS* to the port *Condition* of the rule service, whereas the role *DBMS* represents a basic abstraction of a database client session. Of course, the DBMS role (and the glue process) will undergo major refinements when a condition connector for specific DBMSs is designed.

$$
\begin{array}{l}
\textbf{Connector } \text{ConditionEvaluation} = \\
\quad \textbf{Role } \text{ECAS} = \overline{\mathit{establish!session}} \rightarrow \mathit{Operate} \\
\qquad \textbf{where } \mathit{Operate} = \\
\qquad\quad (\mathit{EvalQuery} \sqcap \overline{\mathit{bot}} \rightarrow \mathit{EvalQuery} \sqcap \overline{\mathit{bot}} \rightarrow \mathit{EvalQuery} \rightarrow \overline{\mathit{eot}});\ \mathit{Operate} \\
\qquad\quad \Box\ (\overline{\mathit{register!conDef}} \rightarrow \mathit{return} \rightarrow \mathit{Operate}) \\
\qquad\quad \Box\ (\overline{\mathit{close}} \rightarrow \S) \\
\qquad\quad \textbf{where } \mathit{EvalQuery} = \\
\qquad\qquad \overline{\mathit{evaluate!(evInst, conDef)}} \rightarrow \mathit{return?result} \\
\quad \textbf{Role } \text{DBMS} = \mathit{open?session} \rightarrow \mathit{DoSession} \\
\qquad \textbf{where } \text{DoSession} = \mathit{newTrans} \rightarrow \mathit{DoSession} \\
\qquad\quad \Box\ \mathit{execQuery?dml} \rightarrow \overline{\mathit{return!result}} \rightarrow \mathit{DoSession} \\
\qquad\quad \Box\ \mathit{commit} \rightarrow \mathit{DoSession}\ \Box\ \mathit{abort} \rightarrow \mathit{DoSession}\ \Box\ \mathit{close?session} \\
\quad \textbf{Glue} = \mathit{ECAS.establish?session} \rightarrow \overline{\mathit{DBMS.open!session}} \rightarrow \mathit{DoQuery} \\
\qquad \textbf{where } \text{DoQuery} = (\mathit{ECAS.bot} \rightarrow \overline{\mathit{DBMS.newTrans}} \rightarrow \mathit{DoQuery}) \\
\qquad\quad \Box\ (\mathit{ECAS.evaluate?(evInst, conDef)} \rightarrow \\
\qquad\quad \overline{\mathit{DBMS.execQuery.f_{dml}(evInst, conDef)}} \rightarrow \mathit{DBMS.return?result} \rightarrow \mathit{DoQuery}) \\
\qquad\quad \Box\ (\mathit{ECAS.eot} \rightarrow (\mathit{DBMS.commit} \sqcap \mathit{DBMS.abort}) \rightarrow \mathit{DoQuery}) \\
\qquad\quad \Box\ (\mathit{ECAS.close} \rightarrow \mathit{DBMS.close} \rightarrow \S)
\end{array}
$$

The function f_{dml} returns the DML statement corresponding to the condition to be evaluated. Due to the similarity to the condition evaluation connector, we leave the specification of a database action execution connector apart.

Finally, we sketch a database event detection connector that permits exclusively asynchronous event signalling as provided in the event notification mechanisms of various commercially available DBMSs. Thus, glue process never blocks the DBMS upon signalisation of a database event but "returns" immediately to the DBMS. In order to be consistent with the database port of the event service, this connector accepts the event *processed?occ* (indicating that all rules for a specific event occurrence are executed) but takes no further measures.

Connector DBEventDetection =
  Role EventService = EventSourceListener
  Role DBMS = $\mathit{open!session} \rightarrow \mathit{Notify}$
    where Notify = $\mathit{defineTrigger?dbEvent} \rightarrow \mathit{Notify}$
      $\Box\ \overline{\mathit{raise!dbEvent}} \rightarrow \mathit{return} \rightarrow \mathit{Notify}$
      $\Box\ \mathit{close?session}$
  Glue = $\mathit{EventService.establish?session} \rightarrow \overline{\mathit{DBMS.open!session}} \rightarrow \mathit{Operate}$
    where Operate =
      $\mathit{EventService.monitor?evDef} \rightarrow \overline{\mathit{DBMS.defineTrigger.f_{trigger}(evDef)}} \rightarrow$
      $\rightarrow \mathit{Notify}$
      $\Box\ \mathit{DBMS.raise?dbEvent} \rightarrow \overline{\mathit{return}} \rightarrow \overline{\mathit{EventService.notify.f_{evDef}(dbEvent)}}$
      $\rightarrow \mathit{Notify}$
      $\mathit{EventService.processed?occ} \rightarrow \mathit{Notify}$
      $\mathit{EventService.close} \rightarrow \overline{\mathit{DBMS.close!session}} \rightarrow \S$

Note that the CSP event *defineTrigger* does not necessarily imply that database events are modeled by triggers, but it stands for the mapping of an event definition to the corresponding DBMS-specific event notification procedures.

5 Conclusion

This paper presents a reference architecture for ECA Systems, featuring services whose functional responsibilities are allocated on the principles of information hiding and separation of concerns. Distinct tasks of active database functionality like rule execution cycle, condition evaluation etc. are placed into distinct architectural entities. For instance, the detection of complex event occurrences is performed completely within the event service so that the choice of another detection mechanism will not affect the other services. Idiosyncrasies of the DBMSs and the computing infrastructure are hidden within the respective connectors which act as virtual machines that provide a common interface to the various systems. Thus the reference architecture promotes a loose coupling of the components which is a major requirement to achieve a high degree of component reuse. The chosen architecture model (i.e., the architecture description language WRIGHT) enables a precise and consistent description of the system, thereby allowing us to reason about the anticipated behaviour of the system.

The description of the reference architecture forms a basis to determine subsequently concrete software components and their implementations. Even though the actual provision of components is dominated by technical features, it is foreseeable that architectural aspects will remain an issue when components and connectors are decomposed into further subcomponents and the design of new elements (e.g., system-specific connectors) must be checked for properties like the local absence of deadlocks. Hence the clear separation of concerns underlying the reference architecture facilitates the design of subsystems because the top-level services have well-defined interfaces that encapsulate changeable aspects from other elements using their facilities.

The architecture description language WRIGHT is for its part a useful vehicle to accompany the subsequent design activities. Hierarchical decomposition of architectural elements can be expressed by means of WRIGHT so that it is feasible to achieve a consistent subsystem design. Accordingly, the reference architecture is transformed into specifications of the system architectures of a specific ECA Systems.

References

[AG94] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference of Software Engineering*, Sorrento, Italy, 1994. IEEE Computer Society Press Washington, DC.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[All97] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[Bla96] J. A. Blakeley. Data Access for the Masses through OLE DB. In *Proc. SIGMOD*, pages 161–172, 1996.

[FGD98] H. Fritschi, S. Gatziu, and K.R. Dittrich. Framboise – an approach to construct active database mechanisms. In *Proc. Seventh International Conference on Information and Knowledge Management (CIKM'98), Washington*, November 1998.

[GD94] A. Geppert and K.R. Dittrich. Constructing the Next 100 Database Management Systems: Like the Handyman or like the Engineer? *SIGMOD Record*, 23:27–33, 1994.

[GD98] A. Geppert and K. R. Dittrich. Bundling: A new Construction Paradigm for Persistent Systems. *Networking and Information Systems Journal*, 1(1), June 1998.

[GKvBF98] S. Gatziu, A. Koschel, G. v. Bültzingsloewen, and H. Fritschi. Unbundling Active Functionality. *SIGMOD Record*, 27(1):41 – 47, March 1998.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. Making the Reuse Business Work. *IEEE Computer*, 30(10):36–42, October 1997.

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), 1992.

[PD97] N. W. Paton and O. Diaz, editors. *Active Rules.* Springer Verlag, 1997.

[Vas94] D. Vaskevitch. Database in Crisis and Transition: A Technical Agenda for the Year 2001. In *Proc. SIGMOD*, Minneapolis, USA, May 1994.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Data Management Systems. Morgan-Kaufman, 1996.