A Component Framework to Construct Active Database Management Systems

DISSERTATION
DER WIRTSCHAFTSWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von

HANS FRITSCHI
von
Flaach, ZH

genehmigt auf Antrag von

PROF. DR. KLAUS R. DITTRICH
PROF. DR. LUTZ H. RICHTER

Dezember 2002

Die Wirtschaftswissenschaftliche Fakultät der Universität Zürich, Lehrbereich Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 4. Dezember 2002*

Der Lehrbereichsvorsteher: Prof. Dr. Martin Glinz

*Datum der Promotionsfeier

The recent notion of components is still very vaguely defined, so vaguely in fact that it is almost advisable to ignore it. N. Wirth [Wir99]

Contents

List of Figures

Acknowledgments

This thesis has been prepared during my employment as a research assistant at the Department of Information Technology of the University of Zurich. I thank to the Ministry of Education of the Kanton of Zurich for funding my position.

I would like to thank to my advisors. I gratefully acknowledge Prof. Dr. Klaus R. Dittrich for his support and helpful comments during the preparation of this thesis. I would also like to thank Prof Dr. Lutz Richter who agreed to provide a review and contributed valuable feedback.

I gratefully acknowledge the contributions of my colleagues in the Database Technology Research Group at the Department of Information Technology. I especially thank to Dr. Stella Gatziu for countless extensive discussions and constructive collaboration. I would also like to thank Dr. Andreas Geppert who carefully reviewed the entire first draft and provided many valuable comments.

Furthermore I would like to thank Dr. Michael Sell who generously accepted to correct the stylistic lapses in earlier versions of this thesis.

Special thanks go to my parents – Lilly and Hans Fritschi – and my friends for their love, understanding and encouragement. They have been a constant source of joy and strength during the ups and downs of the last years.

Abstract

Currently active database systems (ADBMSs) are principally conceived as rather tightly integrated software systems. They are either realized as a part of monolithic DBMSs, or the active database mechanisms are provided as a layer that resides on top of traditional DBMSs. As these modules still can be coupled tightly with the underlying DBMS, the latter approach does not ensure that the respective modules are adapted easily to other DBMSs.

Providing active database mechanisms as individual and customizable database services would therefore open the opportunity to use them in a variety of ways and environments. In that sense, this thesis investigates in the systematic provision of sophisticated active mechanisms in database or database-related environments and proposes an engineering approach to construct active database systems in a cost-effective way.

In a first stage the basic concepts of active database management systems as well as the foundations of (Active) DBMS construction are discussed. These investigations enable the conclusion that the principal approach to be devised in this thesis relies best on decomposing ADBMSs into reusable components that are recombined later on into specific active database services.

In a next step concise meta models to describe the software components and software architectures are elaborated, followed by the definition of specific reuse-oriented software processes to take ADBMSs apart into components and to build active database services out of these components. Subsequently a reference architecture underlying the prospective active database services is devised by applying specialized architecture design techniques. The reference architecture is specified formally by means of an architecture definition language.

Techniques to transform the reference architecture into actual software components are conceived afterwards. The procedure consists of a method to specify components in an implementation-independent way, a technique to generalize them systematically and a process to develop the components with a chosen component infrastructure. In order to recombine the components into a coherent ensemble a method to specify the prospective active database service is devised as well as a schema to classify components and specific tools that assist the software engineer in the assembly of an active database service.

Finally, a prototype has been implemented as a proof of concept.

Zusammenfassung

Heutige Ansätze aktiver Datenbanksysteme (ADBMSe) basieren auf dem Prinzip, dass die aktive Funktionalität ein Teil der DBMS Funktionalität selbst ist. Die aktiven Mechanismen wurden deshalb typischerweise als integraler Bestandteil eines DBMS konzipiert, so dass sie nur in Zusammenhang mit einem konkreten DBMS benutzt werden können. Gelegentlich wurden auch ADBMSe im sog. “Schichtenansatz” entwickelt, einem Vorgehen bei dem die aktiven Mechanismen modular auf einem existierenden passiven DBMS aufgesetzt werden. Auch in diesem Fall ist die Wiederverwendung der aktiven Komponenten in der Praxis recht eingeschränkt, da sie dieselben DBMS-Schnittstellen verwenden die auch jeder “gewöhnlichen” Applikation zur Verfügung steht.

Die Realisierung von aktiven Datenbankmechanismen als eigenständige, adaptierbare Datenbankdienste würde es nun ermöglichen diese in einer Vielzahl von verschieden Umgebungen zu verwenden. In diesem Sinne untersucht die vorliegende Arbeit, in welcher Form komplexe aktive Mechanismen für Datenbanksysteme sowie verwandte Bereiche systematisch konstruiert werden können. Dabei wird ein Verfahren zur kosteneffektiven Realisierung solcher Systeme vorgeschlagen.

In einem ersten Teil werden die Konzepte aktiver Datenbanktechnologie sowie der Konstruktion von Datenbanksystemen analysiert. Daraus wird gefolgert, dass hier der zu entwickelnde Ansatz am zweckmässigsten als Komponentensoftware zu konzipieren ist. Grundsätzlich werden dabei zuerst die ADBMSe in wiederverwendbare Komponenten zergliedert, so dass diese in der Folge zu spezifischen aktiven Datenbankdiensten zusammengefügt werden können.

In einem folgenden Schritt werden einerseits Metamodelle zur präzisen Definition von Softwarekomponenten und -architekturen ausgearbeitet. Andererseits werden spezifische Prozesse zur Zergliederung aktiver Datenbanksysteme in wiederverwendbare Softwarebausteine sowie deren Komposition in kohärente Datenbankdienste definiert. Anschliessend wird eine Referenzarchitektur, welche den zukünftigen aktive Datenbankdiensten zugrunde liegt, entworfen und mittels einer Architekturdefinitionssprache formal spezifiziert.

In der Folge werden Techniken entwickelt, um die in der Referenzarchitektur identifizierten Komponenten (und Subkomponenten) als konkrete Softwarebausteine zu realisieren. Das Verfahren besteht aus einer Methode um die Komponenten in einer technologieunabhängigen aber implementierungsnahen Form zu spezifizieren, einem

Verfahren um sie systematisch zu generalisieren sowie einem Prozess um die Bausteine mittels einer gegebenen Komponenteninfrastruktur zu implementieren.

Zur eigentlichen Konstruktion der aktiven Datenbankdienste werden einerseits eine Methode zur Spezifikation des jeweiligen Systems und ein Schema zur Klassifikation der Komponenten entwickelt. Andererseits werden verschiedene Werkzeuge zur Unterstützung des Konstruktionsprozesses vorgeschlagen.

Schliesslich wurde ein Prototyp zur Validierung der vorgeschlagenen Konzepte implementiert.

Chapter 1

Introduction

Active database management systems (ADBMS) support, beyond the traditional functionality of database management systems (DBMS), the monitoring of specific situations and react automatically in predefined ways when they occur. In the last 15 years, considerable effort has been spent towards understanding ADBMSs and many different approaches have been proposed (for a comprehensive overview see [WC96, Pat98]). However, the richness of these proposals – incorporating features like the monitoring of complex events, various transaction coupling or event consumption modes etc. – contrasts to such an extent with the rather restricted active functionality offered by current commercial DBMSs that one can argue that research in this subject has had little impact on practice. Even though there are important problems which are far from being solved (e.g., concurrency and recovery in the context of active rules, performance of rule processing, methodologies for designing active rulebases and active database applications etc.), we consider the following reasons as decisive why the provision of advanced active database mechanisms has in no way become a topic in the commercial software development.

On the one hand, the absence of industrial-strength ADBMS prototypes results in a lack of comprehensive applications that demonstrate the benefits of active database technology to a broader audience. Consequently, there is still little demand for active database functionality. On the other hand, database management systems – sometimes referred to as "the last major preserve of monolithic closed design" [Vas94] – tend to be loaded with more and more features, implying that these systems grow in complexity and size. This trend obviously leads directly to higher software development costs due to the expanding engineering teams and the increasing testing requirements. In fact, the economy of scale of large, capital-intensive software products is usually quite small, suggesting that DBMS manufacturers might be rather reluctant to commit resources in order to incorporate novel and still somewhat exotic features like active functionality.

The commercial provision of advanced active database mechanisms is furthermore aggravated by the limited standardization of the rule models or languages proposed in various research projects. Even though there is principal conceptual agreement in a number of areas (cf. [DGG95]), there is still little consensus among the sys-

tems. The disagreements are usually attributed to the fact that a choice must be made among a number of reasonable alternatives and each alternative seems to be appropriate for certain scenarios [WC96]. Regarding the rapid evolution in the information systems scenery and its impact on database technology, it is conceivable that this diversity remains an inherent characteristic of active database technology, implying the impossibility to develop all-round, "one size fits all" active database systems. Instead, ADBMSs must be realized in a way that allows them easily to cope with new and unforeseen requirements.

Hence, it makes sense to investigate the *cost-effective* provision of *customizable*, advanced active database mechanisms. Following a general direction that database research is about to take, namely to provide individual database management services that can be used and combined in a variety of ways and in a variety of environments [Vas94, Bla96, GD94c, GD98], we propose the provision of active database mechanisms as an *individual* and *customizable* database service [GKvBF98] that is applicable to arbitrary DBMSs. Customizing an active database service implies that the application engineers can choose among alternative knowledge and rule execution models, and that they can adapt the service for specific application profiles. However, the cost-effective provision of such customizable active database services is a complex task and requires sophisticated construction methods.

1.1 Challenges

Designing a customizable active database service that is applicable for arbitrary DBMSs is different from the development of a specific ADBMS. The important distinction is that the active database service has to cover all relevant concepts in the domain of active database technology, whereas a distinct ADBMS is outlined up front to implement a specific concept.

One must principally combine a variety of approaches to furnish active database behavior into a coherent ensemble that enables the methodical construction of specialized active database services. Thereby one faces a number of challenges.

- *Consolidate the variety of ADBMSs*. Determine what kind of active database behavior shall be provided and which variants shall be included to cover a reasonable spectrum of active database technology.

- *Establish a conception of a configurable active database service*. Identify which aspects of the prospective system are fixed (i.e., occur in all installations) and which are variable, i.e, appear only in specific instances. These decisions imply the design of an appropriate system architecture.

- *Identify the technological approach to implement the system*. One must clarify whether generation or configuration shall be used to build an active database service. Furthermore it is mandatory to acquire a conception of the components

that form a system, e.g., whether they result from a module library or an object-oriented class framework.

- *Determine proper procedures*. Without assistance, beginning with the analysis phase, an expedient – let alone a cost-effective one – construction of active database services is rather unlikely. Thus a productive design activity must be guided by adequate process models.

Finding an answer for each of these issues is not the main challenge, however complex the individual solutions might be. Recall that the most difficult work of software development is not representing the concepts faithfully in a specific computer programming language (i.e., coding) or checking the fidelity of that representation (testing). These activities are accidental parts of software development. Instead, the essence of software development – and the main challenge of this thesis – consists of working out the specification, design and verification of a highly precise and richly detailed set of interlocking concepts [Bro87].

1.2 Outline of the Thesis

This thesis investigates in the systematic provision of sophisticated active mechanisms in database or database-related environments and proposes an engineering approach – named FRAMBOISE[1] [FGD98] – to construct active database systems in a cost-effective way. The conception of FRAMBOISE is the central theme of the thesis which is organized as follows: Chapter 2 sketches the basic concepts of active database management systems. Chapter 3 discusses the foundations of (Active) DBMS construction and elaborates the principal concepts of FRAMBOISE that guide the subsequent investigations. Chapter 4 elaborates concise meta models to describe the software components and architectures that underly FRAMBOISE. Chapter 5 defines specific reuse-oriented software processes to decompose ADBMSs into reusable components and to recombine them into active database services. Chapter 6 develops the reference architecture of the active database service which is specified formally by means of an architecture definition language. The procedure according to which reusable software components are developed in FRAMBOISE is presented in Chapter 7, whereas the ADBMS-specific facilities to assemble a specific active database service are introduced in Chapter 8. Finally, Chapter 9 evaluates the achievements of FRAMBOISE and Chapter 10 concludes the thesis.

1.3 Summary of Thesis Contributions

There is no comprehensive construction theory in the domain of active database technology. Thus contributing a detailed elaboration of the construction system FRAM-

[1] a **FRAM**ework using o**B**ject-**O**r**I**ented technology for **S**upplying active m**E**chanisms

BOISE that conceives the provision of active database facilities as a software engineering process and addresses all relevant phases for the construction is new to this field.

FRAMBOISE enables on the one hand the construction of active database services that interoperate with commercially available, mostly passive, database management systems. These constructs are an effective complement of the trigger facilities or proprietary event notification/detection mechanisms as they are provided by nowadays DBMSs. It is therefore feasible to realize rather comprehensive active database applications.

FRAMBOISE is, however, not exclusively devised to provide active database mechanisms for nowadays still rather monolithically conceived DBMSs. Instead it is feasible to combine FRAMBOISE with a broad range of novel approaches to implement DBMSs, i.e., the so-called *Component DBMSs* (CDBMSs) which have a componentized architecture and allow users to add components.

This leads to the second set of contributions. In order to devise FRAMBOISE, the thesis demonstrates a systematic procedure to work out a precise and detailed set of interlocking concepts to specify, design, verify and classify software architectures, components and their ingredients at various levels of abstraction. As a result FRAMBOISE represents a full-fledged *component framework* [Szy97] to furnish active database systems as CDBMSs. Recalling that it is sometimes argued that component software did not cause a stir in the halls of academia [Mau00] and that component frameworks situated outside the domain of graphical user interface building are still rare [Szy97], this thesis contributes also to the discipline of component based software engineering.

Chapter 2

Active Database Management Systems

Traditional database systems are passive in the sense that they perform operations on the database exclusively on specific (DBMS-) external requests. Active database management systems support, beyond the traditional functionality of database management systems, the monitoring of specific situations and react automatically in predefined ways when they occur. In the last 15 years, considerable effort has been spent towards understanding ADBMSs and many different approaches have been proposed (for a comprehensive overview see [WC96, Pat98]).

This chapter gives a brief survey of the domain of active database technology in order to introduce the proper terminology for the investigations performed in the remainder of the thesis. In Section 2.1 the principal approaches to *specify* and in Section 2.2 the methods to *execute* (re) active behavior are sketched. Subsequently, in Section 2.3 applications of active database systems and in Section 2.4 the principle approaches to provide ADBMSs are discussed. Finally, Section 2.5 concludes the chapter.

2.1 Specification of Reactive Behavior

The reactive behavior of an ADBMS is specified by means of rules that describe the situations to be monitored and the reactions of the system when these situations are encountered. In its general form, rules consist of an event, a condition and an action. Such kind of rules are called *ECA (Event/Condition/Action)-rules* whereby the event and the condition define the respective situation to be monitored. An event is an instantaneous occurrence to which the active database system must react. Events are considered to have no duration, hence they exist as an instant in time.

When an event occurs, the corresponding rules are *triggered*. The rule action is subsequently executed if the condition holds. In other words, the action contains the operations to be performed *when* the event occurred and *if* the condition holds. So-called *rule definition languages* (RDL) provide constructs to specify rules. These RDLs are specific to the respective ADBMS and may therefore vary considerably. In the domain of relational DBMSs established the SQL99 standard [SQL99] among

other things a norm for triggers, whereas these triggers are rather conceived as a minimal subset of ECA rules.

Rules, events, conditions and actions are usually named entities. These names are unique identifiers which may be either manually defined using the rule language or automatically assigned by the active DBMS when rules are stored into the so-called *rulebase*.

We give as example a trading rule from a portfolio management application that monitors and reacts to stock fluctuations, The syntax of this example is derived from the language of the object-oriented ADBMS SAMOS [Gat94].

```
DEFINE RULE LowRisk
ON stock.UpdatePrice
IF (stock.policy = low_risk) and
   (stock.price < stock.initprice * e)
DO stock.Sell
```

Assuming $0 < e < 1$, the rule `LowRisk` ensures that each time *when* a stock price is updated, the bank sells the stocks *if* the stock price decreases with more than $(1 - e) * 100$ percent and the stock policy is `low_risk`,. The event to be monitored is the update of the stock price, named as `stock.UpdatePrice` and defined separately from the rule definition. The condition checks if the policy is `low_risk` (`stock.policy = low_risk`) and if the stock price already has decreased under a given threshold ($stock.price < stock.initprice * e$). If the condition holds then the action `stock.Sell` is executed.

Note that some ADBMSs allow only EA (Event/Action)- or CA (Condition/Action)- rules, i.e., the condition or the event may be either implicit or missing. EA-rules are actually special cases of ECA-rules where the condition always holds. CA-rules are more complex because they imply different semantics to rule processing. In contrast to ECA-rules, where conditions are evaluated only at certain points in time (i.e., when events occur), conditions of CA-rules have to be evaluated continuously.

The elements of rule definitions will be discussed in greater detail, beginning with event definitions (Sec. 2.1.1), proceeding with condition and action definitions (Sec. 2.1.2) end ending with rule execution constraints (Sec. 2.1.3).

2.1.1 Events

An event is a relevant happening that has to be monitored by the active DBMS. It has to be distinguished between the description of an event (also called event definition or specification) and its occurrences. In analogy to programming languages, the former notion corresponds to a type definition whereas the latter represents instances of the defined type.

The design of languages to specify precisely when and under which circumstances rules should be triggered has been important to most of the recent work on ADBMSs.

The two most fundamental event categories are addressed as *primitive* and *composite* events. The former category summarizes events described by atomic occurrences while the latter consist of the (nested) composition of primitive or composite events that are combined through specific operators. Early work on composite event detection was done in the HiPAC project [CBB+88]. A range of active OODBs refined and extended the early results [GJS92, GD94a, CKAK94, CFPT96].

Primitive Events

Primitive events are classified with regard to their origin as *internal* and *external*. Internal events are associated with the access to the database. Examples are:

- *data operation events* occur at the beginning or at the end of a data operation. Thus, the event definition has to specify whether the instances have to occur BEFORE or AFTER the operation has been performed. In relational database systems these may be SQL modification operations, (i.e., `insert, update, delete`), or the retrieval operation `select`, applied on a certain table. In object-oriented database systems, data operations are performed through the invocation of methods that access persistent collections of objects. Therefore, data operation events include the notion of *method events*. Method events may be signalled when persistent objects are created (i.e., the constructor of a class is called) or deleted (i.e., the destructor is called) or when methods modifying or retrieving persistent objects are invoked. For example, let a class `Person` have a method `ModifySalary(x:Integer)`, then the corresponding method event definition is:

  ```
  DEFINE EVENT UpdateSalary
         BEFORE Person.ModifySalary(x:Integer)
  ```

  The event `UpdateSalary` is signalled just before the method `ModifySalary` of class `Person` is invoked.

- *transaction events* occur before or after a transaction operation (begin, abort, commit). For example, the event `Trans` in the following definition is signalled each time the execution of a transaction begins:

  ```
  DEFINE EVENT Trans BOT
  ```

External events are produced by occurrences outside the database, in its environment. The most prominent event types are:

- *time/temporal events* that occur either at absolute time points, e.g., `ON 13.06.98.` or periodically, e.g., `ON EVERY 3 DAY 18:00`.

- *abstract or user-defined events* that arise explicitly within application programs or rule actions. For example, an abstract event defined by

`DEFINE EVENT abstract1` can be explicitly signalled with a command like `RAISE EVENT abstract1`.

- *method events* that are not associated with database changes, occurring at the beginning or at the end of invocations of methods applied on transient objects (i.e., outside the database). For example, an event is raised when a water controlling system signals a change of the water-level. Note that actually the kind of the underlying data model, e.g., relational or object-oriented, determines if the event is defined as a method or operation event.

Composite Events

Primitive events describe elementary occurrences. In order to express more semantics, composite events have been introduced. They are specified by associating the *component events* (or constituent events) by means of specific operators. Component events can be either primitive or composite events. Typical operators to combine events are:

- *Logical operators.* Events can be combined using boolean operators like `AND`, `OR`, etc.

- *Sequence.* A rule can be triggered when two or more events occur in a particular order.

- *Temporal composition.* A rule might be triggered by a combination of temporal and non-temporal events such as "*1 minute after event E*" or "*every hour after the first occurrence of event E*". Temporal composition includes also *interval-based operators* that enable the declaration of time intervals (addressed as *monitoring intervals*) during which specific event occurrences are considered. For instance `[1500 - 1600] E3` means that an occurrence of E3 is only signalled between 3PM and 4 PM.

- *Reductions.* Several occurrences of the same event might be signalled only once or a specific number of times. Typical operators are *closure* that signals only the first occurrence, *history* (or times) that signals every nth occurrence of an event and as a special case the *negation* when an event has not occurred in a specific interval of time.

  The specification of a monitoring interval is mandatory for negations in order to indicate during which period of time the event must not occur. It is, however, conceivable to define a closure without indicating such an interval in order to signal only the very first occurrence.

Quite complex composite events can be defined if an event specification language is based on regular expressions or a context-free grammar. In some systems, the precedence order of operators has to be always specified (e.g., using brackets similar to SAMOS [Gat94]). Another alternative is to provide a predefined order, like in logical

expressions (e.g., conjunction has priority over disjunction, etc.). In this case, a desired order (other than default) may be enforced using brackets. In order to make event specification more precise, event parameter restrictions may be defined as well. They establish whether all component events are signalled for the same object, transaction or user, etc. For example, the event definition `e1 SEQ e2: SAME TRANSACTION` occurs only if e1 and e2 occurred in this order during the same transaction.

Some rule languages like ORCA [Wei96], and Ode [GJ91] shift a part of (or in the case of Ode the entire) selective function of the condition to the event. The aim is to avoid unnecessary rule triggering. For example, ORCA provides the event specification with a where part:

```
ON UPDATE TO employee.salary
                      WHERE age < 30 and salary > 4000
```

Here the `WHERE` clause is considered as a part of the event definition instead of being a condition.

Event Parameters

Event parameters establish the connection between the database state at the point in time when an event occurs and the rule behavior by passing information about the actual database state to the condition and action. There are two kinds of parameters: fixed and variable.

Fixed parameters are automatically produced by the system:

- *timestamp* is the time point assigned by the system to the event occurrence,
- *oid* represents the object identifier of the object for which a method is called or a specific tuple of an object-relational database,
- *trans_id* is the identifier of the *triggering transaction* (i.e., the transaction during which the event has occurred),
- *user_id* is the identifier of the user who started the transaction in which the event has occurred.

The last three parameters are usually only meaningful for primitive events because the components of composite events may originate from different objects, transactions or users objects and therefore have no common `oid`, `trans_id` or `user_id`. It is however possible to restrict the components of complex events on the same object, transaction or user so that the definition of the corresponding event is feasible.

Variable parameters are parameters that have to be defined during rule specification. Usually, they depend on the event type. For example, method events have the parameters of the method they are defined on. With few exceptions (e.g., disjunctions), composite events are attributed with all parameters of their components [Gat94].

The syntax of rule languages may be extended with special keywords to query a state of the database prior to the current one. Such keywords may be `inserted`, `deleted`, `updated`, `old`, `current`. When used in conditions and actions these keywords address the state of the database before insertion, deletion or modification. For example, existing systems support the query of the state before the execution of the actual transaction (e.g., Chimera [CFPT96]), the state after the last evaluation of the rule (e.g., Chimera, Starburst [WCL91]) or the state before the occurrence of the rule event (e.g., Oracle [Ora92], POSTGRES [SK91], NAOS [CCS94]).

2.1.2 Conditions and Actions

Conditions specify what has to be checked in order to ensure whether the action of a triggered rule is executed at all. If the result of the condition evaluation is `true`, the condition is *satisfied*. A condition may be:

- *a database query*. The condition might be simply specified as a query, using the database system's query language. For example, in relational database systems a condition may be anything corresponding to an SQL `WHERE` clause. A condition evaluates to `true` if the query produces a non-empty set and to `false` otherwise. The result of the query may be passed on to the action.

  Some systems restrict the expressiveness of conditions for reasons of performance. In this case, the predicate may be defined by only using a restricted query language. For example, comparison operations are allowed but aggregate functions and joins are not.

- *one or more application procedure calls or method invocations*. Boolean procedures or methods may be conditions as well. Similar to queries, procedures or methods may also return sets of data.

The *action* specifies the reactive behavior of a given rule. Actions principally may perform arbitrary operations, such as data modification and retrieval in the database, transaction operations like `commit` or `abort`, method invocations (in object-oriented DBMSs), procedure calls and rule operations.

The flexibility in the specification of actions and conditions depends on the used language which is also referred to as *condition/action* language. Usually the DML of the underlying DBMS is used. Examples of condition/action languages range from specialized database languages (e.g., SQL for Starburst [WCL91], POSTQUEL for POSTGRES [SK91] and Ariel [Han96]) to general-purpose programming languages with persistent extension (e.g., O++ for Ode [GJ91], Smalltalk for HiPAC [CBB+88] or a persistent extensions of C++ for SAMOS [Gat94]). A rule language obviously improves its expressive power if the applied condition/action language tends towards a general purpose programming language. In this case, not only sequence and selection are supported, but also programming constructs such as abstraction and iteration. This

allows conditions to contain any procedure or method invocation that returns a boolean value and actions to include any executable program. In this case, conditions and actions are more complex than those specified using simple rule languages based on database languages.

2.1.3 Execution Constraints

Some rule languages allow the specification of execution constraints such as priorities and coupling modes. Furthermore rule execution can be constrained by *enabling* and *disabling* (also called activate and deactivate) rules in order to "switch" them temporarily on or off respectively. Disabled rules remain in the system but cannot be triggered or selected for execution.

Rule operations may be used in rule definitions in order to perform the rule manipulation. For example, within the action part of a rule, another rule may be enabled/disabled, and so on.

2.2 Rule Execution

A so-called *execution model* specifies how rules are treated at run time. Even though the details of an ADBMS' execution model are closely related to system specific aspects such as the data model or the transaction manager, rule execution can be subdivided into a number of steps that basically apply for all ADBMSs [Pat98]. These steps are depicted in Figure 2.1 and will be discussed in the subsequent paragraphs.

2.2.1 Event Signalling

When primitive events *occur* in an event source, they are *detected* by the associated primitive event detector. A *composite event detector* checks whether the primitive event occurrences contribute to composite events. The composite event detection is carried out until the composite event detector reaches a final state and no more composite events can be detected. Existing work proposes different techniques for composite event detection: extended finite state machines (Ode, [GJS92]), event graphs (Sentinel [CKAK94], NAOS [CC96]) and colored Petri Nets (SAMOS [GD94b]). New approaches deal with distributed concurrent detection models [TGD97]. The result of detection is the *signalling* of events which means that timestamps are assigned to events and the active system is informed about event occurrences. Various alternatives exist to calculate the timestamps:

- *The system assigns unique timestamps to events.* Even if physical events occur simultaneously, the system registers them at different points in time. Thus, only one event is signalled at a point in time. Primitive events are principally timestamped before simultaneously occurring composite events.

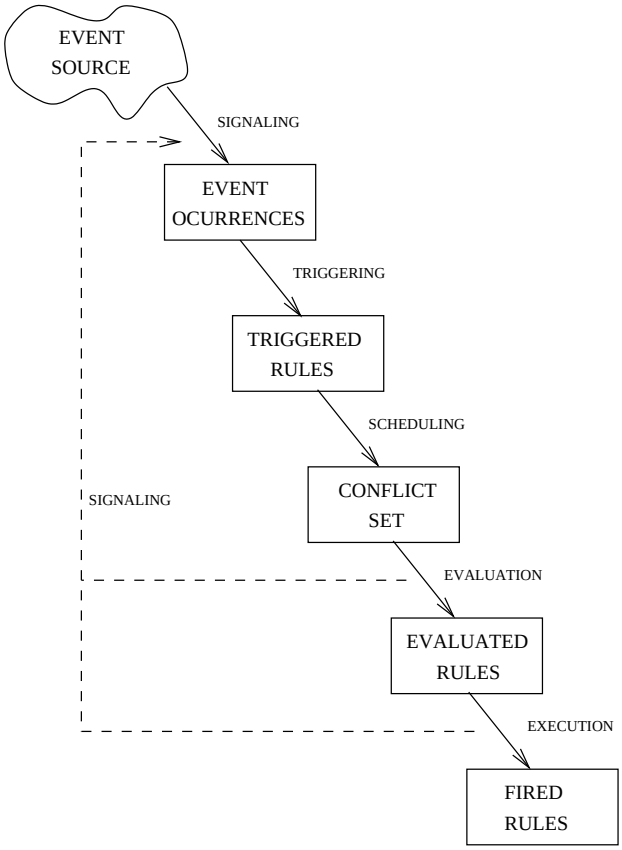

Figure 2.1: Principal steps which take place during rule execution [Pat98]

- *Multiple primitive events can get the same timestamp if they occur simultaneously*. Composite events get the timestamp of the last occurred constituent event(s). Inherently, there may be (primitive and composite) events with the same timestamp.

Combinations of these variants are possible. For example, primitive events have unique timestamps but composite have the same timestamp as the last occurred primitive. The option chosen for timestamp assignment plays an important role for rule processing because it establishes the order in which rules are triggered (cf Sec. 2.2.2) which in turn influences the selection order of rule execution (cf. Sec. 2.2.3).

The *event consumption mode* [CKAK94, FT95] determines whether events retain their capability to participate in rule triggering after their associated rules have been processed and at which point in time they lose this capability (i.e., events are consumed). There are basically two alternatives:

- Event consumption is performed in relation to rule triggering. Primitive events are consumed when their rules are triggered. [CKAK94] identifies four choices for the consumption of constituent events: *recent* (e.g., ACOOD [BL93]), *chronicle* (e.g. SAMOS [Gat94]), *continuous* (e.g., NAOS [CC96]) and *cumulative*. The most prominent option is chronicle, i.e., for the detection of a composite event, the oldest eligible occurrence of each constituent event is considered. A specific event occurrence may participate in one or more composite events. After a composite event occurred and the appropriate rule is triggered, the constituent event occurrences are consumed. This is not necessarily the case for recent and continuous consumption where some constituent event occurrences may be reused for the detection of further composite events. In contrast, cumulative consumption requires the accumulation of the occurrences of constituent events until all necessary constituent events have occurred. Then, the composite event is detected and all occurrences (including the accumulated ones) are consumed.

- Consumption is only affected by condition evaluation. According to [FT95], rule events remain ”pending” as long as conditions are false (e.g., Ariel [Han96] and NAOS [CCS94]). The events are subsequently consumed when conditions hold.

All event occurrences are registered in the *event history* (or event log), which is a list sorted according to timestamps. The history begins as soon as the first event type is defined. The history usually lasts over many sessions and over several transactions. Its purpose is to support composite event detection (which relies on past event occurrences), to enable recovery in the case of a failure and to keep a record of all event occurrences for later analysis.

2.2.2 Rule Triggering

Once the event occurrences are signalled the ADBMS *triggers* the rules defined for the respective events. The association of a rule with its event occurrence forms a *rule instantiation*. The nature of the relationship between events and the rules they trigger is partially captured by the *transition granularity*. This indicates whether the relationship between event occurrences and rule instantiations is 1:1 or many:1. The transition granularity *tuple* implies that a single event occurrence triggers a single rule (e.g. updating a set of tuples in a relational DBMS will trigger a rule for each modified tuple). The transition granularity *set* implies that a collection of event occurrences triggers a single rule (e.g., deleting a set of tuples triggers one rule instance).

Another feature that influences the relationship between event occurrences and the rules they trigger is the *net effect policy*, which indicates whether the net effect of the update events rather than each individual event occurrence shall be considered. For example, creating a tuple in a relational DBMS and modifying the tuple immediately afterwards can be considered as identical to the creation of the modified tuple. Net effect policies are not generally meaningful when rules are processed immediately, because the event occurrences must beforehand be collected in order to determine the net effect. Specification of net effects is for instance supported in the ADBMSs Starburst [WCL91], Chimera [CFPT96], Ariel [Han96] and A-RDL [WC96].

2.2.3 Rule Scheduling

The scheduling phase of rule execution determines what happens when several rules are triggered at the same time. Active database systems that have to cope with large quantities of data efficiently in a context where deterministic behaviour is highly desirable, usually select the next rule to be fired by means of static rule priorities. The priority of a rule specifies the execution order of a rule in relation to other simultaneously triggered rules. Priorities may be absolute or relative. In the former case, absolute values are attributed for each rule to impose an execution order on the rules. In the latter case, priorities specify ordering for pairs of rules. Rule instantiations that have the same priority are grouped to so-called *rule conflict sets*. Hence there remains the task to select the rule instances in a conflict set, which is referred to as *conflict resolution*. In ADBMSs there are basically two approaches to resolve conflicts:

- *deterministic* conflict resolution, which is applicable if a total order among rules exist. Rules are for example chosen for execution based on their timestamps or on static properties such as the time of rule creation etc.

- *nondeterministic* conflict resolution is applied if only a partial order is given or no ordering exists at all. In that case rule instantiations are chosen arbitrarily.

The second issue of the scheduling phase is the number of rules to be fired. The following options are typically found in ADBMSs:

- Firing *all* rule instantiations of a conflict set *sequentially*. This approach is usually applied for rules supporting integrity maintenance.

- Firing *all* rule instantiations of a conflict set in *parallel* in order to enable more efficient rule processing. In that case no conflict resolution takes place. This method demands sophisticated concurrency control mechanisms in order to prevent that rule instantiations interfere with each other. For instance, concurrent threads in the same triggering transaction that stem from parallel rule execution require some kind of subtransactions.

- Firing all instantiations of a specific rule before any other rules are considered. This approach is preferably used in the domain of expert systems but is also conceivable for specific ADBMS applications.

After scheduling the rule instantiations, rule evaluation and execution takes place.

2.2.4 Rule Evaluation and Execution

In the *evaluation* phase the conditions of the triggered rules are evaluated. If the condition holds, the rule fires and may be executed in the rule execution phase. *Rule execution*, sometimes called *rule firing*, represents the phase where the action of the selected rule is executed.

Rule execution – and in some systems also rule evaluation (e.g., the invocation of a method in an object-oriented ADBMS) – can lead to the signalling of further event occurrences which in turn trigger subsequent rules. This is addressed as *cascaded* rule firing.

Depending on the *atomicity* of conditions and actions [FT95] the evaluation of a condition, respectively the execution of an action, are either *atomic* or *interruptable*. In the latter case the respective operation can be suspended in order to process other eventually triggered rules. Atomic rule execution implies that the current condition evaluation or action execution must terminate before further rule instantiations may be processed. The atomicity of conditions and actions constrains the so-called *cycle policy* [WC96, Pat98] which determines how cascaded rule firing is treated. A *recursive cycle policy* implies that the current action execution (or condition evaluation) is suspended in order to process the rule cascade spawned by this operation, i.e this cascade overrules the actual conflict set. Obviously a recursive cycle policy requires interruptable actions and conditions. An *iterative cycle policy* means that rules instantiations triggered in a cascade are simply added to the conflict set without interrupting the current operation and are later on scheduled for processing as any other rule instantiation. Atomic condition evaluation and action execution imply an iterative cycle policy. Note that the outcome of rule processing with an iterative cycle policy is different from that of a recursive cycle policy relying on interruptable operations.

2.2.5 Coupling Modes

The phases of rule execution discussed so far are not necessarily executed contiguously but depend on the so-called *coupling modes* [Day88] which are pairs of values `(x, y)` associated with each rule. The value `x` couples event signalling and condition evaluation of a rule whereas `y` couples condition evaluation and action execution. The coupling modes specify the processing of a rule along the two dimensions, *time* and *transaction*, i.e., `(x, y)` describes the time and/or transaction relationship between a rule event and the evaluation of a rule condition, respectively between the evaluation of a condition and the execution of an action. Possible coupling modes are `immediate`, `deferred` and `decoupled` [FT95].

- *immediate* coupling. The condition is evaluated as soon as the operation that produced the triggering event(s) terminates. The same applies to the second coupling mode. In particular, the action is immediately executed after the condition evaluation. The exact point in time depends on what is considered to be ”noninterruptable update unit”: operation, method call, etc. `immediate` also means that the condition evaluation or the action execution are performed within the same transaction as the previous step (i.e., event triggering, respectively condition evaluation). This implies that an abort caused by the rule produces an abort of the transaction.

- *deferred coupling*. The evaluation of the condition and/or the execution of the action are delayed until some other occurrence,usually the attempt of the transaction to commit, takes place. There are also approaches that delay rule processing until specific *rule assertion points* are signalled.

- *decoupled* coupling (also called detached). The evaluation of the condition, respectively the execution of the action takes place in a separate transaction related to the event, respectively to the condition. In this case, the condition evaluation and action execution are not under the responsibility of the active part of the system, but are part of the concurrency control system of the underlying DBMS. The decoupled mode can be subdivided into *dependent* and *independent* decoupled. In the former case, the separate transaction is only started when the original transaction commits, in the latter, it is started independently of the original transaction.

2.3 Applications of ADBMSs

The notion of *passive database applications* refers to applications that do not make use of any active features even though the underlying DBMS might offer them. *Active applications* are not only based on DBMSs with active capabilities, but make actually use of these capabilities [SKD95]. Active applications can be classified into the following categories [PD98]:

- *Database System Extensions*. Active database mechanisms are applied to provide other facilities of a DBMS. For example, ECA rules have been used to support integrity constraints, materialized views, transaction models, version management etc.

- *Closed Database Applications*. This category includes the use of active database functionality to implement application related tasks without actually referring to external devices or systems. Hence closed database applications rely exclusively on database and complex events whereby the condition/action language is an extension of the respective DML.

- *Open Database Applications* use the ADBMS in conjunction with monitoring devices to record and respond to situations outside the database. This category forms a superset of closed database applications.

The range of active applications that can be provided by a specific ADBMS directly depends from the *expressiveness* of its rule language, i.e., the variety and number of supported constructs as well as the number of elements and constructs provided by the condition/action language. Experience has shown [KDS98] that an ADBMS should includes at least composite operators, coupling modes and rule operations as well as a condition/action language that provides all elements of a general purpose programming language in order to be considered as expressive.

Depending on the complexity of a given application, the design of a rulebase may evolve into a complex development activity. ADBMSs should therefore provide mechanisms and tools to support rule development such as RDL compilers, specialized editors and browsers etc.

A challenging problem in the design of active applications is to predict how the rules will behave at runtime. Due to the various coupling modes, cycle policies and the occurrence of cascaded rule firing, the interleaving of rule behaviour can become quite complex even with simple rulebases. The order in which rules are fired may not be immediately obvious from the rulebase and the rule priorities. It is therefore important to furnish an ADBMS with tools to support the automatic analysis on sets of rules in order to predict certain aspects of the rule behavior. For example, analysis techniques can be used to determine whether a set of rules is guaranteed to terminate or whether it will behave deterministically. When these aspects cannot be guaranteed, rule analysis tools may help to isolate the rules responsible for the problem. Even though the necessity of such design assistances to enable an effective use of ADBMS technology is undisputed, the nature of these means is not yet fully understood. Rule development is consequently still a research topic. A set of tools and techniques to support rule development for ADBMSs has been proposed in [Vad99, Dia98].

2.4 ADBMS Architectures

The functional components of ADBMSs have been identified for the first time for the object-oriented ADBMS HiPAC [MD89]. These elements are:

Event Detectors Detect events and signal them to the rule manager.

Rule Manager Maps events to rule firings and rule firings to transactions.

Condition Evaluator Evaluates rule conditions.

The Architecture of HiPAC is sketched in Figure 2.2. Note that the event detectors are incorporated in the *Transaction Manager* and the *Object Manager* that provide nested transactions and object-oriented data management respectively. The latter two building blocks are not exclusively part of an *active* DBMS but passive DBMSs incorporate components with similar functionality whereby tan object manager is only applicable for DBMSs with an object-oriented data model.

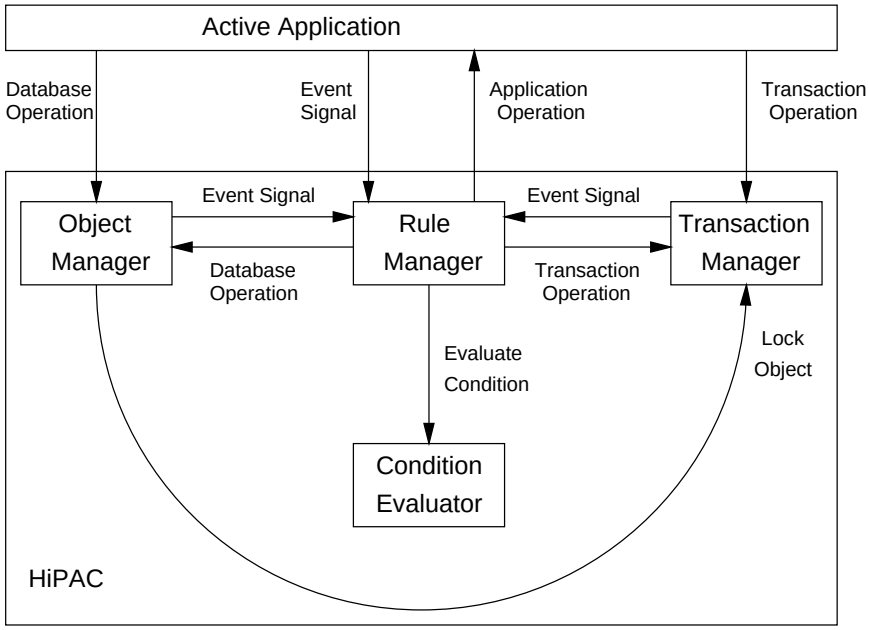

Figure 2.2: Functional Components of the HiPAC Architecture

Subsequent work dedicated to the architecture of ADBMSs has addressed mainly the system architectures of research prototypes whereby these proposals were typically driven by implementation specific aspects. Thus the various system architectures vary considerably. Nevertheless it is nowadays widely agreed that ADBMSs in order to provide the active capabilities in addition to full DBMS features, are composed out of standard DBMS facilities of the database system (e.g., transaction manager) and

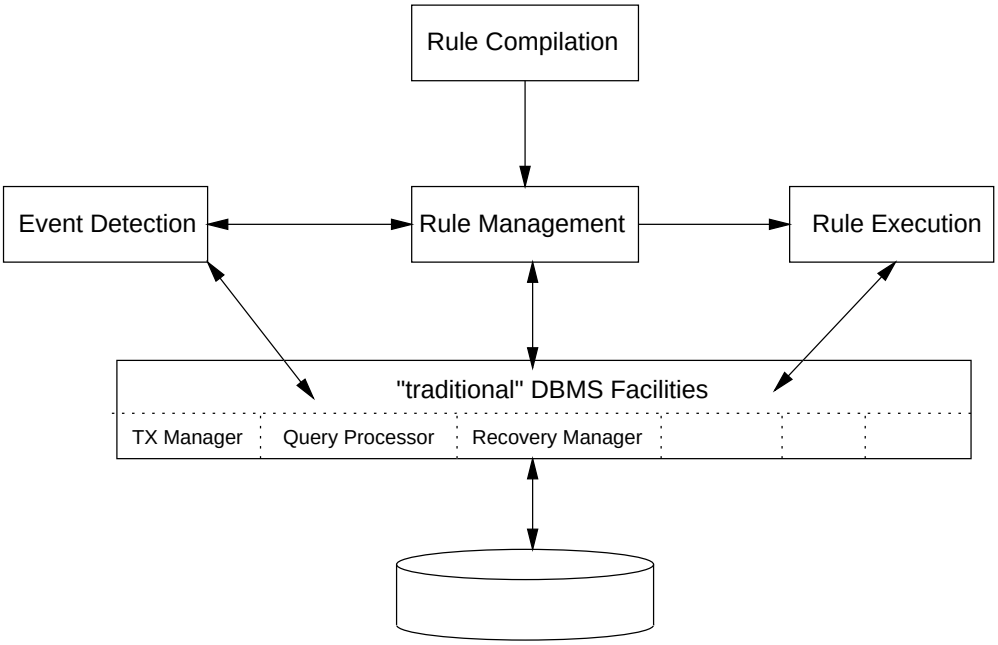

Figure 2.3: Principal Subsystems Providing Active Database Functionality

building blocks providing active features such as event detectors, a rule manager and a rule executor as illustrated in Figure 2.3. The basic issues of ADBMS architectures are furthermore discussed in [Buc98].

In order to implement ADBMSs there have basically been applied two approaches:

- *Layered approach.* All active database components are built as a layer on top of a conventional passive database system. The advantage is that existing database systems can be converted into active systems without internal modification. For example, SAMOS [GGD+95b] is built on top of a commercial available DBMS, ObjectStore [Obj93]. The disadvantage of the layered approach is that it has potential for poor performance due to the communication overhead between the "active" layer and the actual DBMS. Furthermore certain features (e.g., specific coupling modes) that require access to the DBMS subsystem may not be supported.

- *Built-in approach.* In this case, all active database components become part of the database system itself. This architecture can be achieved by modifying an existing passive database system (e.g., NAOS [CCS94] is an extension of the object-oriented DBMS O2), by using a database system toolkit for conventional features (e.g., REACH [BZBW95] relies on the Open OODB core mechanisms [WBT92]) or by building all components from scratch. The latter approach requires a substantial effort and has to our very best knowledge not been applied so far.

Active database functionality can also be provided to some extent by a technique called *query modification* [SB99a]. In this method – also called *compiled approach* – DML commands are modified at compile time to include effects of active database rules. This approach has the distinct advantage that event monitoring and rule processing are unnecessary which reduces the complexity of implementation task and is claimed to improve the overall performance of the ADBMS [WC96]. However, this approach is only applicable for restricted rule and execution models. For instance all events must be detectable through the compiler so that the provision of temporal events and complex composite events is not possible. Furthermore cascaded rule firing implies that query modification must be applied recursively to provide the DML statements representing the cascade. Since rule termination cannot be guaranteed at compiletime there is the risk that the compilation phase may continue indefinitely even though the cascade would terminate at run-time [WC96].

Finally, in order to render active database functionality useful at all, adequate development tools have to be provided as discussed in Section 2.3.

2.5 Conclusion

Most of the techniques discussed in this chapter have been realized in research environments. However, active database technology scarcely influenced practice. Nowadays (object-) relational DBMS provide triggers which are actually simple ECA rules, restricted to database operation events that might be signalled for every affected tuple or once for a set-oriented operation. As far as it is known, none of the popular commercially available object-relational DBMS provides any form of complex events. The Condition/Action language is usually restricted to DML-statements and proprietary database programming languages. Trigger execution typically enables immediate and sometimes deferred coupling. Cascading rule execution usually aborts the triggering transaction when a cascade visits a relation twice even though the cascade might actually terminate. Furthermore the number of triggers per table is often limited.

Concerning object-oriented DBMSs the situation is even worse. As far as it is known there is no commercially available object database system that provides more than simple event notification mechanisms. Applying these raw event notification facilities burdens various tasks to the application developers such as programming demon processes to perform blocking reads of the event queues, event signals are to be numbered, user-defined information is to be packed in strings that are part of the notification etc. It has been shown in [KHS94, KDS98] that the limitations of expressiveness and execution models encountered in commercially available DBMS impede the development of active applications of industrial-strength and contributes likewise to the little popularity of active database mechanisms.

Chapter 3

Foundations of ADBMS Construction

Conceiving a construction system that enables the provision of all potential software systems of the respective domain is an unrealistic assumption. It is therefore of paramount importance to determine carefully what kind of constructs shall be furnished by means of the prospective construction system. This applies all the more for the domain of active database technology with its inherent variety of approaches (cf. Chap. 2). In that sense, this chapter contributes the establishment of a scope of an active database construction system that ensures on the one hand a broad applicability and makes on the other hand the system design to a realistic endeavour.

The chapter starts with a survey of previous approaches to construct DBMSs in general (Sec. 3.1), because constructing ADBMSs belongs in its essence to the field of DBMS construction. Subsequently in Section 3.2 other proposals to construct active database mechanisms are presented. It is obvious that software reuse is a central theme of any (cost-effective) construction system. Thus the principal aspects of software reuse are discussed in Section 3.3.

The chapter sketches subsequently in Section 3.4 the assumptions and requirements underlying the construction system to be devised in this thesis. This construction system is henceforth referred to as FRAMBOISE[1]. Afterwards in Section 3.5 the software engineering principles needed to provide FRAMBOISE are settled. Finally, Section 3.6 concludes the chapter.

3.1 Database Construction

This section looks at related work in a broader sense by presenting previous approaches to systematic and cost-effective DBMSs construction. In section 3.1.1 the so-called *extensible DBMSs* are introduced. They gave the rationale to actual *DBMS construction systems* which are discussed in section 3.1.2. In Section 3.1.3 we introduce a novel conception of DBMSs, i.e, the so called *component database systems*. This survey is

[1] a **FRAM**ework using o**B**ject-**O**r**I**ented technology for **S**upplying active m**E**chanisms

concluded by discussing in Section 3.1.4 how the DBMS construction approaches are reflected in commercially available products.

3.1.1 Extensible DBMSs

The term *extensible database systems* [CH90] names a direction of database research that investigated DBMSs that were intended to support the addition of new features like new data types, functions, complex objects, storage techniques and access methods and to incorporate recent advances in DBMS technology in a timely manner.

Extensibility was desired at all abstraction levels. At the top level (i.e., the associative or user level) the support of abstract data types (i.e., adding new data types and new operations on such data), as well as the extension of the query languages with new set operators such as transitive closure was provided. At a lower level, extensible DBMSs should enable to enhance the query processing with new execution strategies, including new implementations of operations and new ways of combining these operators. Examples of these facilities are the addition of new highly efficient join methods which in turn make use of likewise new index structures to support queries over spatial data. Finally, extensible DBMSs should allow for the definition of new data storage methods, thus making it possible to accommodate new storage media such as optical disks.

The various approaches are principally distinguished according to the style of construction and the construction phases they support or which aspects (e.g., transaction management) they address. They are typically classified into the following categories:

Kernel Approaches offer a more or less fixed functionality, which implements the lower layers of a DBMS. Examples of a kernel approach are DASDBS [PSS+87] or more recently BESS [BP95].

Customizable DBMSs can be modified/extended at specific, well-defined places in the system (e.g., attachments in STARBURST [HCL+90]). In this approach, the range of the constructible DBMS is rather restricted (e.g., due to a fixed data model).

Toolkit Systems (e.g., EXODUS [CDG+90]) offer reusable components which are selected, eventually modified and plugged together by the database implementor (DBI) in order to achieve a coherent DBMS or DBMS subsystem. Sometimes these approaches are also addressed as *configurable approaches*.

Generators or Transformation Systems (e.g., GENESIS [BBG+88], the EXODUS optimizer generator [GD87], DMC (Data Model Compiler [MBH+86]) take descriptions of an aspect (or an entire DBMS) as input and (semi-) automatically create corresponding implementations.

Due to the variety of different approaches, the term extensible database systems is sometimes considered as misleading [Gep94]. Many of the proposed systems are extensible like any other software system: Further components are added on top or internal components are replaced. Other systems are not really extended, but are reused in their entirety and are plugged into enclosing, new systems. In other cases, it is not the (database) system which is extended, but the library or repository of specifications or reusable modules. Summarizing, the expression *DBMS construction approaches* is preferred over extensible DBMSs, because it is not always a full-fledged DBMS that is extended.

3.1.2 Database Construction Methods

The DBMS construction approaches discussed in the previous section suffer from several shortcomings:

- Building customizable systems usually requires advanced programming skills.
- The selection of alternative techniques is not supported and left entirely to the DBI. Likewise, toolkit approaches burden the DBI with the definition of an appropriate architecture and the integration of the chosen implementation modules.
- Extensible DBMSs mainly considered the implementation of single DBMS-components and techniques for specific DBMS-tasks such as query optimization or transaction management. Neither new features can be easily integrated nor is there a choice among supported alternative realization techniques.
- Guaranteeing the integrity of a DBMS and its data in the face of extensions was far from being solved.

In fact, the systematic construction of DBMSs as a whole has scarcely been considered. Thus, to reduce both the amount and complexity of the work required to construct DBMSs, [GD94c] points out the necessity of an actual DBMS construction method based on proper software engineering methods. The following requirements that should be satisfied by a DBMS construction method were identified and analyzed:

1. *Architecture.* A generic and adaptable architecture model is required that is applicable for a broad range of significantly different DBMSs.
2. *Knowledge Representation.* Knowledge about database technology (e.g., on alternative realization techniques for a specific task or on experiences of previous designs) has to be expressed.
3. *Design for Reuse.* Design for reuse has to be enforced, i.e., decomposition of techniques and components into easily reusable artifacts.

4. *Specification-Based Design*. The construction method should be based on specification techniques.

5. *Design Completion and Integration*. Based on the architecture design and the requirement specifications, the construction method has to support completion of the design and the retrieval of adequate artifacts. Based on the architecture framework and the selected techniques, the construction method should supply the integration of required modules (be they generated or composed). This assembly results in an operational DBMS (at least partially).

These aspects were first elaborated in the KIDS project [Gep94, GSD97] that aimed at the development of a DBMS construction approach by defining specification-based approaches strongly relying on software reuse.

3.1.3 Component Database Systems

The approaches discussed so far proposed some modularization of DBMSs, but the respective decompositions were just a means to an end in order to enable an efficient construction process. The resulting DBMSs were still built as rather closed systems. The emerging pervasiveness of personal and internet computing induced a further reconsideration of the monolithic conception of DBMSs, in order to cope with the upcoming demands on database technology. In this section the visions formulated so far are sketched and the state of the art is discussed.

A Novel Conception of Database Technology

Due to the availability of personal computing resource (e.g., desktops and notebooks) a vast amount of critical information necessary to conduct day-to-day business is stored outside the traditional production corporate databases [Bla96]. Instead information is found in file systems, index-sequential files (e.g., Btreive), personal databases (e.g., Access, Paradox) and productivity tools (e.g., spreadsheets, project management, email). Transferring such data from their original storage system into a DBMS in order to benefit from database technology such as declarative query languages, transactions and security, leads to redundancies and is often too expensive. Conversely, applications often want to exploit the advantages of database technology not just when accessing data within a DBMS, but also when accessing data from any other information container.

Thus, [Vas94] emphasizes the necessity to build DBMSs in a way that they can interoperate with a variety of data sources. Rich data models are required and database languages should be decoupled from the DBMS, because a developer simply wants the use of a database manager without being forced to pick a particular language, object model or development framework. In that sense, this technical agenda conceives DBMSs as *component DBMSs*, i.e., they will be constructed in the long term by putting

together individual components, as it is no longer feasible to satisfy all upcoming requirements that might be posed to a DBMS by a single monolithic system.

The World Wide Web and the Internet will lever this trend still further. Since they enable the cooperation of heterogeneous software components in an almost unlimited variety of ways, an increased demand to integrate reliable (i.e., DBMS-like), distributed and heterogeneous storage facilities is predictable as well as their necessity to cope with enormous workloads (in both terms, data to be managed and users) [Vas95]. The latter aspect applies also for active database mechanisms. Recalling traditional examples for the application of ADBMSs such as air reservation systems, stock markets or retail systems, it is easy to conceive that an ADBMS must scale to support millions of user preferences encoded as ECA rules if the application is accessible via the World Wide Web. Rule systems that are realized either be modifying the rule executor or by query modifications, however, are not appropriate to support environments that have a large number of rules.

DBMS Construction Reconsidered

[GD98] elaborates on the principal *engineering aspects* of component-oriented database construction. DBMSs are constructed by *(re) bundling* them, i.e., putting together preexisting, reusable and compatible components in a systematic way. The components to be bundled 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*. 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 DBMSs.

3.1.4 State of the Art

Nowadays commercial DBMSs increasingly support abstract data types as they have been devised for extensible DBMSs. They provide means to define so-called *extensible data types* which are packaged with their associated functions, operators and access modules into specific modules which are in turn called by the respective DBMSs. These modules are called *DataBlade modules* in Illustra/Informix [SB99b] (Oracle uses the term *cartridge* [ora97] and IBM's DB2 *extender* [IBM95] to describe a similar concept).

OLE DB [Bla96] defines an open, extensible collection of interfaces that factor and encapsulate orthogonal, reusable portions of DBMS functionality. These interfaces define the boundaries of DBMS components such as record containers, query processors

and transaction coordinators that enable uniform transactional access to diverse information sources. Hence a DBMS becomes a conglomerate of cooperating components that consume and produce data through a uniform set of interfaces. It is feasible to interchange notifications among OLE DB components and clients. These notifications are considered as a basic mechanism on which to implement active database behavior.

The Common Object Request Broker Architecture (CORBA) [Obj97] specifies a number of *object services* (i.e., collections of system level services packaged as components [OHE96]) that correspond to specific database services, e.g., a concurrency control and a transaction service, a persistence or a query service.

3.2 Construction of Active Database Mechanisms

In this section related work in a narrower sense is surveyed by discussing other approaches to construct active database mechanisms (Sec. 3.2.1 to 3.2.4) and first ideas to unbundle ADBMSs in Section 3.2.5.

Note that the implementation of the various ADBMSs is skipped in this context. Since they were usually implemented as ADBMS research prototypes to verify specific active database concepts, database construction techniques were not specially taken into account. Nevertheless some of these research prototypes anticipated the necessity to build ADBMSs at least partially out of preexisting parts e.g., as layer on top of a passive DBMS [GGD+95b] or by implementing the ADBMS as an extension of a database construction toolkit [CKTB95, BZBW95].

3.2.1 AIDE

[Jas94] is a three-layered toolbox to construct active information systems (AISs). The lowest layer is an interface layer that allows the integration of foreign systems into the prospective AIS by means of active abstract data types (AADTs) encapsulating several event types. The top layers provide means to put the AADTs together i.e., various specification and data manipulation languages as well as tools for the development of active applications. AIDE proposes comprehensive event handling mechanisms (e.g., a rich set of complex events) but to our very best knowledge it is not outlined to offer different rule execution and knowledge models.

3.2.2 Active-Design

[BFL+97] sketches the initial ideas of a toolkit, called Active-Design, that is used to provide sophisticated active rule processing mechanisms that run on top of "light-weight" active DBMSs supporting restricted active functionality. The toolkit is conceived as a set of reusable building blocks that are combined to implement the various components of the so-called *Active Monitors* (AMs). In contrast to our approach, the

design of the toolkit is function-oriented rather than architecture-driven. The modules forming the toolkit are coupled along a layered architecture where each module uses the interfaces provided by the modules(s) right below and beneath. However, the functional decomposition of an AM is not directly reflected in the toolkit architecture. Thus, the cohesion of the modules is rather weak as the procedures in the various modules are primarily logically associated [Som92].

3.2.3 Amalgame/H2O

[BDD+95] proposes a CORBA-compliant toolkit that includes among other things a so-called *activeness service* that provides a basis for constructing so-called *active modules*. These are – in terms of CORBA – application objects that use the activeness service in the context of a particular application and incorporate a rulebase, a rule execution model and optionally a local persistent store. Active modules are specified by means of the H2O language and subsequently generated by composing them of components of the Amalgame toolkit. The toolkit is up front designed for its application in a CORBA environment and is not specifically outlined to interoperate with DBMSs. Thus, the active modules can potentially interact with arbitrary data sources by applying for instance the CORBA event service.

3.2.4 TriggerMan

[HK97] is the first proposal and implementation of a software system (ATP) that processes triggers asynchronously after transactions have committed in the source database. An ATP is designed to be able to gather updates from a wide variety of sources (by providing an extensible data source mechanism) incorporates sophisticated methods to evaluate complex (i.e., multitable) conditions. As far as it is known, TriggerMan is not outlined to process complex events. Even though an ATP is extensible in several ways and there are potentially various forms of such ATPs, TriggerMan is not designed as an actual construction system including tools and methods that support the development of ATPs themselves.

3.2.5 Unbundling of ADBMSs

The first ideas to separate active database functionality from the DBMS in order to provide so-called *activity components* were discussed in [GKvBF98, KGvBF98]. A series of consecutive unbundling steps sketch various generic architectural proposals that can be used as a starter-kit to initiate the unbundling of active database mechanisms for a specific purpose.

In order to cope with the scalability of ADBMSs when large numbers of triggers must be processed, Stonebraker and Brown [SB99a] propose an approach similar to the TriggerMan project (cf. Sec. 3.2) with a stand-alone rule execution engine that is associated to the DBMS execution engine (cf Figure 3.1). This engine receives

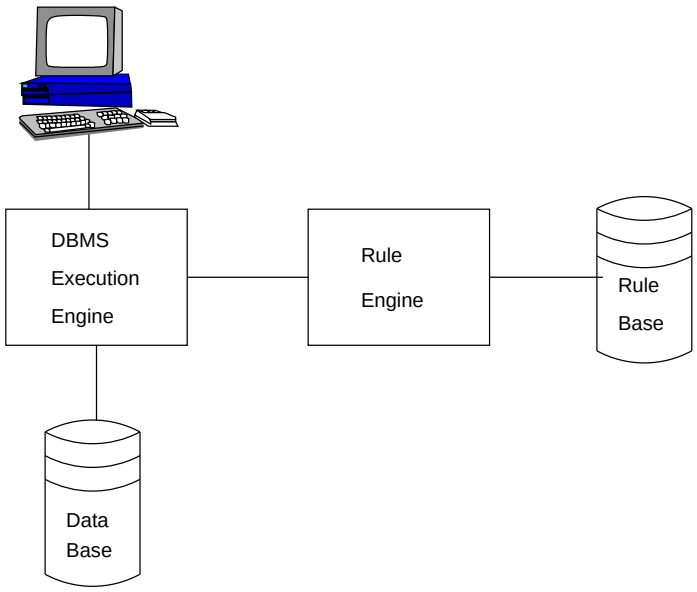

Figure 3.1: Proposal for a Scalable Rule System [SB99a]

each DBMS update and passes it through a highly optimized data structure e.g., a discrimination network, that encodes active rules.

3.3 Software Reuse

It is now widely recognized that reuse techniques improve the quality of the software as well as they decrease the time of the software development and the time-to-market of the final products [Rin97]. Thus it is obvious that software reuse must be a cornerstone of any construction system aiming at cost-effectiveness.

Software reuse is defined as the process of creating software systems from existing software rather than building them from scratch [Kru92]. It can be considered from six perspectives [PD93]:

Substance defines the *essence* of reused items, i.e., ideas, concepts, software building blocks, skills etc.

Scope defines the form and extent of reuse, i.e., *domain-specific* versus *general-purpose* reuse, *internal reuse* (within the same software system) versus external reuse, *small-scale* (procedures) versus *large-scale* (entire software systems) reuse.

Mode defines how reuse is conducted, i.e., *ad-hoc* reuse versus *systematic* reuse which is based on a formal process model.

Technique defines the approach that is used to implement reuse, i.e., compositional reuse (by assembling software building blocks) versus generative reuse (reuse of a tool that generates reuse by generating programs on account of a specification).

Intention defines how elements will be reused, e.g., *as-is reuse* versus *reuse by adaptation*.

Product defines what is reused, e.g., specification, design, architectures, source code, documentation etc. The entirety of reusable products is often addressed as reusable *artifacts*.

The above *facets of reuse* summarize the *technical aspects* of software reuse. In order to implement a successful reuse program, the following technical problems must be solved in a satisfying manner [Kru92]:

Selection As libraries of reusable artifacts can be very large, the retrieval of the artifacts suited for reuse in a concrete situation must be supported.

Integration Typically, it is not possible to build a new and coherent software system by simply assembling the artifacts identified as suited for the respective task. Instead, the selected artifacts need to be integrated implying that support for this activity is necessary.

Specialization Often there are no artifacts available that fulfill exactly the needs of a specific situation. Typically, an artifact that fits relatively best is reused by specializing it for the current context. Consequently, this customization activity needs to be supported and the modified artifacts should be added to the respective library.

Abstraction In order to enable the aforementioned activities, reusable artifacts must be represented in an abstract way. This abstraction must allow to conceive easily the semantics of an artifact and if the latter is appropriate for the system under construction.

The technical aspects addressed so far are mandatory prerequisites for any successful reuse but they are not sufficient to make reuse happen. Systematic reuse requires also long-term, top down management support because years of investment may be required before it pays off, legal issues may have to be considered and changes in organizational fundings and management structures may be necessary [FI94].

3.4 Assumptions and Requirements of our Work

In this section, we identify the basic notions of our prospective ADBMS construction system FRAMBOISE based on the following assumptions:

- FRAMBOISE shall enable the construction of active database mechanisms as an *individual (active) database service* that interoperates with a specific database management system.
- There may be several DBMS-specific instances of this active database service.
- We establish the ECA rule paradigm as basic concept of our active database service, because this is more generally applicable than EA and CA rules (which can also be implemented as general cases of an ECA rules). We refer subsequently to this active database service as *ECA System* (ECAS).
- FRAMBOISE shall be applicable to arbitrary DBMSs. Hence there may be several DBMS-specific instances of ECA Systems.
- ECA Systems shall be *customizable* by enabling the application engineers to choose among alternative rule and execution models.

The basic features of ECA Systems are presented in Section 3.4.1 in greater detail. Subsequently the alternative rule and execution models provided by FRAMBOISE are specified (Sec 3.4.2). Finally, the requirements of a cost-effective construction process is addressed in Section 3.4.3.

3.4.1 Features of ECA Systems

ECA Systems are basically active database mechanisms that interoperate with database management systems. Thus they represent database middleware that is able to process active database rules, according to specific knowledge and rule execution models. These models are determined by specific users of FRAMBOISE – addressed as *ADBMS-implementors* (ADBIs) – according to their needs.

The overall system formed by an ECAS and the respective DBMS shall provide the functionality of an active database management system as it is outlined in the "Active Database Management System Manifesto" [DGG95]. On the one hand, the overall system must preserve the concepts of the passive DBMS such as "passive" modeling facilities, query languages, multi-user access, recovery etc. On the other hand ECASs must support the following features:

- *ECA-Systems shall support the definition and management of ECA-rules* i.e, ECASs have to provide means to define events, conditions and actions.
- *ECAS shall have an execution model* i.e., they detect event occurrences, evaluate conditions and execute actions. These operations will often not be performed by an ECAS itself but by the DBMS or external application programs (e.g., the DBMS signals an update or evaluates a query representing a condition). The ECAS performs rule execution by processing the events and invoking the respective operations according to its rule execution model. Thus, an ECAS must

have a well-defined execution semantics, i.e., event detection, signalling and consumption must be well-defined such as the coupling modes that determine when, how and on what database states conditions are evaluated and actions executed. Finally, conflict resolution must either be pre-defined or user-definable.

- As an optional feature, the Manifesto proposes that an ADBMS should represent information on ECA-rules in terms of its data model. As for an ECAS, this means that it must be possible to *maintain a rulebase of an ECAS using the functionality of its associated DBMS*. This should, however, not be a mandatory feature, i.e., an ECAS may have its own rulebase storage facility.

- An ECAS shall support a rule definition language that is specialized for the respective knowledge and rule execution model. Furthermore, an ECAS comes along with an *ADBMS programming environment* including tools like rule browser, rule designer, debugger, trace facilities etc. in order to assist the user of an operational ECAS to define and maintain a rule schema.

- *ECASs should be tunable*. Therefore an ECAS must provide "hooks" for the application of performance tuning tools to measure eventual bottlenecks.

For the time being, we rule the provision of distributed active database mechanisms out. Hence an ECAS interacts exclusively with one DBMS server.

3.4.2 Active Database Functionality covered by FRAMBOISE

The rule models or languages proposed in various research projects are only marginally standardized. Even though there is principal conceptual agreement in a number of areas (cf. [DGG95]), there is still little consensus among the systems. The disagreements are usually attributed to the fact that a choice must be made among a number of reasonable alternatives and each alternative seems to be appropriate for certain scenarios [WC96]. Regarding the rapid evolution in the information systems scenery and its impact on database technology, it is conceivable that this diversity remains an inherent characteristic of active database technology, implying the impossibility to develop all-round, "one size fits all" active database systems.

Thus FRAMBOISE must enable the construction of ECAS that cover the dimensions of active database behaviour listed in table 3.1. This table is derived from [PDW+93] that introduces a number of dimensions of active rule system behaviour in order to survey differences between various proposals. The symbol $\subset$ is used to indicate that the particular dimension can take more than one of the values given, whereas $\in$ indicates a list of alternatives. For a detailed explanation of the various elements listed in table 3.1 cf. [PDW+93].

Event	Type $\subset \{Primitive, Composite\}$ Source $\subset \{Structure\ Operation, Behaviour\ Invocation, Transaction, Clock, Error, External\}$ Granularity $\subset \{Instance, Collection\}$ Role $\in \{Mandatory\}$
Condition	Mode $\subset \{Immediate, Deferred, Detached\}$ Role $\in \{Mandatory\}$
Action	Mode $\subset \{Immediate, Deferred, Detached\ Dependent, Detached\ Independent\}$ Options $\subset \{Update\ Db, Abort, Do\ Instead, Update\ Rules, Inform, External\ Call\}$
Execution Model	Transition Granularity $\subset \{Tuple, Set\}$ Net-effect policy $\in \{Yes, No\}$ Cycle Policy $\in \{Iterative, Recursive\}$ Priorities $\in \{Dynamic, Numerical, Relative, None\}$ Scheduling $\in \{All\ Parallel, All\ Sequential, Saturation\}$
Management	Operations $\subset \{Activate, Deactivate\}$ Description $\subset \{Programming\ Language, Objects, Extended\ Query\ Language\}$ Adaptability $\in \{Compile\ Time, Run\ Time\}$ Data Model $\in \{Relational, Extended\ Relational, Object\text{-}Oriented\}$

Table 3.1: Active Database Behaviour covered by FRAMBOISE

3.4.3 Cost Effectiveness

ECA Systems are not an end in itself, but provide active database services which are in turn used by other software systems. Hence the provision of the ECAS will be only one task of many to be accomplished in a software development project implying the particular importance of a cost-effective construction.

It is principally justifiable that the provision of an ECAS requires a certain effort, because it is not an everyday task. Cost-efficiency relies therefore primarily on a construction process that ensures a proper quality of the final product. Recalling the complexity of active database mechanisms it is easy to conceive that the construction of an ECAS compensates economically on mid to long-terms over ad hoc implementations that fake some active database functionality. It requires quite some expertise in ADBMS technology to get such "hand woven" implementations right and one risks a considerable maintenance effort to enhance these software systems continuously with additional features (e.g., specific coupling modes or complex events) that belong to the standard repertoire of ADBMS technology but are difficult to provide.

Hence cost-efficiency will primarily be provided by means of software quality. Nevertheless, the objective is that time and human resources to build an ECAS are minimal.

3.5 Software Engineering Principles underlying FRAMBOISE

It is a truism that software reuse does not come for free but must be fostered thoroughly and has an associated overhead cost and effort. It is therefore indispensable to establish up front according to which (reuse-oriented) software engineering principles a software system shall be built. Accordingly this section identifies the adequate techniques to provide a cost-effective active database construction system. These methods are in the remainder of this thesis applied to furnish FRAMBOISE.

Section 3.5.1 specifies which of the reuse facets introduced in Section 3.3 are suitable for the purpose of this thesis. Based on these principles, Section 3.5.2 settles the construction paradigm that is suited for our purpose. Finally, Section 3.5.3 sets the process according to which FRAMBOISE is furnished.

3.5.1 The Reuse Facets of FRAMBOISE

The construction system devised in this thesis is dedicated to investigate database construction techniques, therefore the focus is on *technical aspects* of software reuse.

Scope FRAMBOISE is a construction system specialized to provide active database mechanisms. Thus it has a domain-specific (vertical) reuse scope, whereby large-scale external reuse (i.e., among multiple ECAS) is primarily envisioned.

Mode FRAMBOISE aims at the systematic construction of ECAS and must therefore be guided by concise process models.

Technique FRAMBOISE is conceived as a so-called *transformation system* that processes a specification of an ECAS and produces (semi-)automatically an ECAS out of prefabricated components or eventually a skeleton where missing components must be filled into. In order to achieve flexibility, components shall be designed in a way that they can also be used without applying the generator.

This hybrid method where a generative approach is built on top of a composition--based one is a sensible compromise. Generation-based systems are usually domain-specific and have the advantage that the reused patterns can be designed and implemented carefully by experienced programmers. Considering that FRAMBOISE is domain-specific and that there are patterns that appear in all ECASs (e.g., every ECAS implements a rule execution cycle) the adoption of a strictly generative approach seems attractive. However, generative approaches cannot be applied easily in all situations, because they are often too general or too specific for applications under consideration. Since FRAMBOISE must provide for interaction with an unlimited range of DBMSs and applications, it is impractical to rely exclusively on generative techniques. On the other hand, composition-based approaches are generally applicable to a wider variety of applications, but, compositional reuse requires larger efforts to master the selection, integration etc. problems addressed in section 3.3. This can be mitigated to some extent with a generator that assembles a skeleton of an application, indicates missing parts and eventually proposes good candidates to be inserted.

Intention Reuse that relies exclusively on shrink-wrapped components is inflexible and usually too restrictive [PD94]. Thus it is illusory that ECAS can always be assembled out of preexisting components, even though there will be a substantial number of components that are generic enough to be reused "as-is" throughout most ECAS. Hence, FRAMBOISE must provide for reuse by adaptation.

Product FRAMBOISE should not restrict software reuse to source code and their resulting components, 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 ECAS might be reused in later constructions (e.g., designs, specifications, test cases etc.).

Substance FRAMBOISE is principally intended to reuse special skills and concepts about ADBMSs. Focusing on the reuse of components as reuse technique does not exclude the reuse of ideas and concepts, because the reuse of a component automatically means the reuse of the ideas and concepts that are built into the component. This applies in particular for large-scale components let alone entire architectures that intrinsically incorporate much ADBMS knowledge.

3.5.2 ECAS Construction Based on Bundling

By discussing the reuse facets of FRAMBOISE, the reusable software were identified as one of the cornerstones of an active database construction system. Hence FRAMBOISE corresponds in its essence with the conception outlined in [GD98] (cf. Sec. 3.1.4) namely to construct DBMSs by *rebundling* them out of prefabricated components. Thus many of the ideas and notions formulated there are also applicable to devise a system to construct ADBMSs. [GD98] identifies two levels of abstraction for (un-) bundling:

- The *meta level* determines the necessary concepts and models for bundling, e.g., it defines constructs to model un- and rebundled systems.
- The *instance level* deals with concrete components and new systems (processes at this level are referred to as *instance (un)bundling processes*).

Software engineering researchers and practitioners agree that the quality of the resulting software product depends to a high degree on whether a proper software process model is defined and enacted during a construction. In that sense, [GD98] describes the provision of a bundling oriented construction system by means of an informal *meta bundling process* that indicates how the necessary concepts and models for bundling shall be determined. This meta process is described as follows:

1. *Perform domain analysis*. Domain analysis is defined as the process of identifying and organizing knowledge about some class of problems – the problem domain – to support the description and solution of those problems [PDA91]. In this phase, the domain of bundable systems is analyzed through domain analysis techniques [PDA91]. This step clarifies what the required functionality should cover in detail. A further important step in domain analysis identifies *fixed* and *variable* aspects, because some elements of bundable systems are variable in the sense that they are not contained in every system or that different variants can be pursued for them. Some parts will be fixed for all rebundable systems because the same functionality is required by all systems in exactly the same way.

2. *Define a software architecture model*. In this phase an architecture model capable to define the software architecture of the rebundled systems is established. The most important ingredients of such an architectural model define the notion of component as basic units and relationship types between these components.

3. *Define the unbundling and rebundling processes*. The *instance unbundling process* yields one or more functional aspects identified in the domain analysis which can be represented in the architecture model. An instance unbundling process spans all activities from the domain analysis of the aspects to be unbundled up to the provision of the components which are integrated in adequate repositories. Conversely, the *instance rebundling process* starts with a requirement

analysis, provides an operational system built out of components and completes with the maintenance of the respective system.

4. *Perform an unbundling process for an initial set of systems (if monolithic systems are available) or identify bundable components*. Typically, several unbundling processes are performed, because it is not sensible to unbundle a complex system in a single step. Instead, the various aspects to be unbundled are considered progressively.

The provision of an ECAS is closely related to the construction of *one* aspect of a DBMS according to this conception. In that sense FRAMBOISE is a construction system to provide ADBMSs by (re) bundling ECASs out of prefabricated components and integrating them with the other aspects of DBMSs – which may in turn be rebundled entities.

Since bundling-oriented database construction is still in its infancy, aspects concerning component software technology have not yet been considered in depth. Component software technology, however, is considered as one of the most searched and at the same time least understood topics in the software engineering domain [Szy97]. Thus we emphasize the component engineering aspects in our work and consider bundable DBMSs (including ADBMSs) subsequently as specific component software.

3.5.3 The FRAMBOISE Meta Bundling Process

In order to furnish FRAMBOISE we need a process that guides the activities to determine the necessary concepts and models. In accordance with the terminology established in [GD98], we refer to this process as Meta Bundling Process. The elements listed in the meta bundling process we discussed above, are an adequate basis to define such a process for our purpose. However, this list must be enhanced by an activity to establish a proper *component model* in order to clarify our perception of a reusable software component. Since architecture models rely on the notion of components it is sensible to define the component and the architecture model in the same phase of the meta bundling process.

We achieve the meta bundling process for the FRAMBOISE project as shown in Figure 3.2. The provision of FRAMBOISE will proceed according to this meta process.

3.6 Conclusion

This chapter settled the FRAMBOISE construction system to provide active database mechanisms that interoperate with a principally unlimited range of database management systems. The final products constructed by means of FRAMBOISE – referred to as ECA-Systems (ECASs) – represent database middleware that is able to process ECA rules, according to specific configurable knowledge and rule execution models.

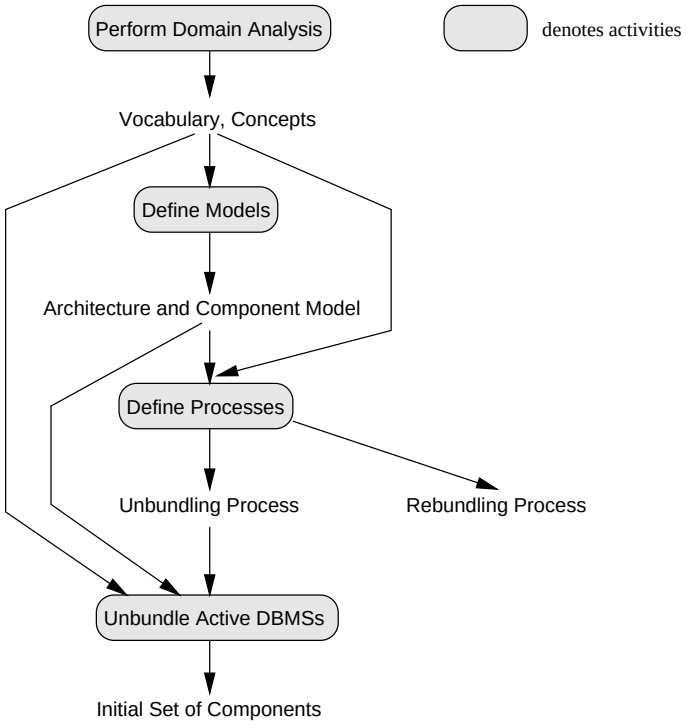

Figure 3.2: The Meta Bundling Process of the FRAMBOISE Project

Furthermore the basic technology identified, namely ECASs are built out of prefabricated reusable software components. Thus FRAMBOISE is conceived as a component framework that is provided according to the meta process depicted in Figure 3.2. The initial domain analysis has principally been accomplished in chapter 2 so that the next chapters proceed with the definition of the component and architecture model (Chap. 4) as well as with the instance bundling processes (Chap. 5).

Chapter 4

Software Models

This chapter investigates in component and architecture models that are adequate for an active database construction system. Whereas there is nowadays a wide agreement about program constructs like classes, modules, procedures etc. such an unanimity did not emerge yet concerning software components and architectures. Establishing a concise perception of software components and architecture is therefore an indispensable prerequisite to successfully devise any component based construction system.

Since there exists a large amount of publications about component software and software architectures, this chapter is to a large extent a discussion of existing literature and approaches. The actual contribution of the chapter is that it settles in the midst of this babel a proper component and architecture model that enable the designer as well as the users of an active database construction system to grasp the essence of its software artifacts. The component model is described in Section 4.1 and the architecture model in Section 4.2.

4.1 The Component Model

A component model defines basically the following aspects:

- the perception of a reusable software component,
- how components are connected to form the overall system, and
- the information that must be stored in a repository together with the respective component.

The main characteristics of software components are examined in Section 4.1.1. They are delimited afterwards in Section 4.1.2 from the concepts of object-oriented and modular programming. Subsequently techniques to form composite systems out of components (Sec. 4.1.3) are examined and component management issues are discussed in Section 4.1.4. Finally, the component model underlying this thesis is derived in Section 4.1.5.

4.1.1 The Characteristics of Software Components

There are basically two aspects that form the essence of a software component, i.e, what kind of software building block it is and how it is delimited from its environment. We discuss first the notion of a software component in general and address subsequently component interfaces, followed by the presentation of a technique called *reification* that enables components to present meta information. Finally we introduce various abstraction levels to expose component internals to component developers and users.

The Notion of a Software Component

The discussion about reusable software components began at the same time (i.e., in 1968) with the emerging of the term software engineering [NR68]. Even though the number of articles and trivia published on this subject grew exponentially over the last years, no generally agreed conception of component software has been established.

The literature offers definitions of software components – subsequently simply referred to as components – that dissent on several aspects. First, there is no agreement on the granularity of components which may range from simple macros or templates [Jac93, Sam97] to complex subsystems (so-called *megamodules* [WWC92]). Sometimes it remains unclear how components are used [Boo87] or the system composition time is emphasized [NT95]. Moreover, components are either perceived as an abstract notion used as a design philosophy independent from any concern to reuse existing components or they can be seen as off-the-shelf building blocks used to design and implement component software [BW98]. Probably the most rigorous perception has been established in [Szy97]

> A software component is a [binary] unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Deploying a component denotes that it can be installed in the target system without modification, whereas being a *unit* of independent deployment signifies that a component cannot be deployed partially. Sometimes, deployed components are also referred to as a *component instances*. Since components are units of deployment, they must be distributed with all immutable resources that are not listed as explicit context dependencies (e.g., images or other frozen medias used by the component for presentation issues). Note that mutable resources have not to come along with the component because they belong to the respective component instances.

Component Interfaces

In order to become applicable by third-parties, components must be sufficiently self-contained and they must be separated clearly from their environment and from other

components. Thus it is mandatory that components interact with their environment through well-defined *interfaces*. They are a means to make the context dependencies of a component explicit. For instance, an approach proposed in [BBK+97] distinguishes between so-called *provided* and *required* interfaces. The former specify the services or capabilities a component offers to other components whereas the latter specify services that a component must receive from other components to perform their functionality. There are approaches where components may have multiple interfaces. This is useful to structure complex functionality or to provide backward compatibility.

In their most basic form interfaces are a set of named operations that can be invoked by clients of the respective component. In a wider sense, however, component interfaces are regarded as contracts [Mey92a] between clients of an interface and the provider of an implementation of the interface. A particularly popular contractual specification method is to capture the two sides of a contract by specifying pre- and postconditions for an operation by means of a so-called *Hoare Triple* (i.e., a triple `{precondition} operation {postcondition}`). Contracts are a simple idea but have subtle implications because revised implementations must continue to obey the postconditions and may still rely on the preconditions. Actually, further revisions of a component may only weaken preconditions or strengthen postconditions.

Formal contracts (e.g., such as those relying on Hoare triples) are, however, not sufficient to provide usable components. On the one hand, attempting to formalize everything can easily lead to components that are nearly impossible to grasp and are therefore quite useless. Thus it is important not to overspecify component interfaces. On the other hand, components must also obey requirements such as availability, mean time between failures, throughput, data safety, capacity etc. It is, however, not yet clear how such requirements might be specified best in contracts. An interesting approach is to specify time and space complexity bounds in a contract to ensure certain performance requirements in a platform independent way [Szy97].

Reification

Static interface definitions are a valuable source of information about components, but they are not sufficient to support development tools like browsers and debuggers or to generate self documenting components and finally to support versioning and licensing procedures.

Further meta information is preferably gained directly from the components if they are able to *reify* meta information. This implies that the respective component implementation technique must be able to preserve information available at compile time (e.g., class or method signatures) for inspection at runtime. Making this information available to a system is called *reification* [KdRB91]. Metadata gathered by means of reification is always accurate but it provides the risk that customers might decompilate a component and thus get insights into internals that a component manufacturer does not want to provide.

Component Abstractions

Besides delimiting components and establishing a contract to invoke services, interfaces are also *abstractions* of components that hide implementation details to varying degrees. Generally, the following levels of abstraction are distinguished:

Blackbox Abstraction No details beyond the interface and its specification are made known to clients.

Graybox Abstraction Controlled fractions of component implementations are revealed. The notion is considered as dubious because partially revealed implementations could be seen as parts of the specification.

Glassbox Abstraction All internals of a component can be seen from outside, but must not be changed. Hence the implementation of components can be studied to understand what the abstraction does, but the components themselves are used like black-boxes.

Whitebox Abstraction The interface may still enforce encapsulation and limit what clients can do even though implementation inheritance – if applied – enables considerable interference. However, implementation details of a whitebox are fully exposed and can be adapted.

Software components should preferably be applicable without knowing internals of the component i.e., they are conceived for *blackbox reuse*. The corresponding blackbox abstraction is provided by the aforementioned ability of the components to reify. However, blackbox abstraction is not sufficient because it is impossible to prevent situations where a proper component cannot be found and one is obliged to implement one from scratch. Thus component repositories often provide whitebox abstractions, whereas some components might be exposed exclusively in a glassbox abstraction in order to prevent unsolicited modifications.

4.1.2 Components versus Objects and Modules

Modular and object-oriented programming techniques form an effective code reuse technique. In fact, classical software libraries can be and are sold in binary form and up to now they are the most successful form of reusable software building blocks. However, the concept of objects and modules differ from the strict component notion established above.

Object-Orientation

Components and objects are typically related but they do not depend on each other. It is feasible to provide components which are built on top of procedural or functional

approaches or even by means of assembler languages. The other way round object-oriented languages obviously can be applied to build fully operational programs or libraries instead of components.

The distinction between components and object-oriented concepts shows itself regarding that on the one hand a component typically consists of several classes but a class is on the other hand not necessarily confined to one single component.

Modules

Since component technology implies modular solutions, component software relies on modularization techniques. The opposite, however, does not strictly apply i.e., not every modularization yields components in the strict sense. It is for instance common practice to bundle thematically related operations (e.g., traditional mathematic libraries) in distinct modules according to a design guideline indicating that each module should be simple enough that its implementation can be fully understood [BCK98]. Such a modularization is sensible but such modules hardly can be perceived as components.

In that sense, module concepts actually are rather focused on implementation of software building blocks than on the provision of composite systems. Nevertheless, modules – unlike classes – are sometimes seen as minimal components. However, they differ from the above notion of full-fledged components, because module concepts typically do not support the delivery of immutable persistent resources with modules (apart from hard-wired constants) [Szy97].

4.1.3 Composition Techniques

Components are deployed independently but they are never deployed in isolation. They are rather deployed to cooperate with other components to form operational composite software systems (i.e., ECAS or ADBMSs respectively). Effective component software relies on techniques that enable developers to integrate components in novel constellations and possibly unforeseen deployment areas. Furthermore, flexible upgrade and replacement of system components must be feasible, regardless whether they are developed in-house, supplied by third-parties or purchased off-the-shelf.

Such a compositional style of development requires methods that go beyond the traditional approaches where systems are finally built by linking software modules together. We discuss in the next section how components can be connected and outline subsequently the provisions of component infrastructures with respect to technical aspects of component integration.

Connection-Oriented Programming

The activity of putting components together in order to build a system is usually referred to as *Connection-Oriented Programming* [Szy97] in order to emphasize that

component instances are connected in order to collaborate. We discuss in this section the principal connection mechanism and how components are usually assembled.

Connection Mechanisms Component interoperation means that components invoke services from other components and perform in turn operations for other components. It is principally feasible to implement component interoperation by applying traditional caller-driven programming techniques. However, such components are rather tightly coupled because of the explicit reference that a caller has to maintain to the callee. The asymmetry of connection primitives (i.e., procedure calls or method invocations) reinforces this effect when components must hold mutual references.

Message-oriented connections allow to replace such rather statically chained call dependencies by indirections that can be configured arbitrarily at runtime. That means that parts of a system interact with each other by exchanging message entities that "travel" from a sender to one or several receivers. Such connection mechanisms are increasingly factored out as separate messaging (or event) services (e.g., MQSeries, TIBCO etc.). Hence it is feasible to transport messages indirectly by sending them to a distribution service that propagates them to the receivers interested in the respective message. The latter either register with such a service or the service uses a multicasting or broadcasting strategy to locate potentially interested receivers.

Component Assembly The connection mechanisms are the basic ingredients of the so-called *glue* [NL97] i.e., the software that connects specific component instances. Gluing is often performed by applying lightweight programming techniques like *scripting*. Scripts are essentially small – typically procedural or functional – programs that intercept events on their path from a source to a sink and trigger special actions. This technique closely resembles shell programming practiced in UNIX environments. Scripting approaches have been successfully integrated in visual development environments like Microsoft's Visual Basic or Borland's Delphi.

Component Infrastructures

Component infrastructures furnish the technical elements to *wire* [Szy97] software components. Basically, this addresses the aspects of component interface specification required to ensure binary compatibility of components, the handling of references when they leave their local process, the location of services and the handling of component evolution.

The binary compatibility of components is an important issue because it is unthinkable to ask all vendors of dependent components – and the vendors of components depending on these components etc. – to recompile and redistribute within any reasonable time. Unfortunately on the binary level there are principally no conventions that go beyond the invocation of procedures. Since contemporary operating systems and their system libraries have procedural interfaces, there was no need for operating

system manufacturers to define method calling conventions so far. Consequently, current object invocations do not follow standard calling conventions with the result that code compiled using different compilers might not interoperate, even if implemented in the same language.

There emerged a number of approaches providing 'component wiring' standards that try to capture their share of the emerging component markets. Nowadays most prominent approaches are the CORBA-centered standards emerging mainly from the world of enterprise computing [Obj97], the COM-centered standards [Rog97] which evolved out of Microsoft's dominance of the desktop area and finally Sun's Java-centered standards [GJS96].

Component Frameworks

Mere component infrastructures are not sufficient to effectively build composed systems. The feasibility to build arbitrary systems out of components is mostly hampered by the so-called *architectural mismatch* [BCK98]. This describes a disagreement between the assumptions embodied in separately developed components that manifests itself architecturally. For instance two components dissent about which one invokes the other. Avoiding architectural mismatches requires environments that set the conditions how the components are to be connected to form composite systems. Such deployment areas are usually settled by *component frameworks*. In the broadest sense, component frameworks are component-oriented development environments [NL97] that consist of a dedicated and focused architecture, a set of interfaces and the rules that govern how the components 'plugged into' the framework may interact. Component frameworks usually provide an implementation that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the framework. For instance, a component framework might enforce some ordering on event multicasts and thus exclude entire classes of subtle errors caused by glitches or races that could otherwise occur. The implementation of the component framework and those of the participating components, however, remain separate.

The most prominent example of a component framework is OpenDoc [FM96], a compound document architecture that has originally been developed by Apple and IBM. Component frameworks outside the domain of graphical user interface building are nowadays still rare [Szy97], a fact that has been confirmed recently in [Mau00]. One of these seldom encountered examples is JBed/RTOS [Mic], a realtime operating system consisting of components, which has been developed by Oberon Microsystems.

Framework Hierarchies

Component frameworks are usually targeted for specific domains and it is nowadays accepted that single component frameworks that fit for every purpose are illusory [Szy97]. Thus building complex composite systems may require the incorporation of several component frameworks, which implies in turn the necessity of an *interoper-*

ation design for the participating component frameworks. Such interoperation designs can be devised as frameworks of their own with component frameworks as plug-ins. Szyperski proposes a three-tiered meta architecture where the first two tiers are formed by the components (first) and the component frameworks (second) [Szy97]. Frameworks situated in the first tier are also referred to as *first order frameworks* whereas those in the second tier are classified as *second-order frameworks*. The third tier incorporates the component framework interoperation design. However, as current approaches only provide single component frameworks it is not yet clear how the third tier should be built.

4.1.4 Component Management

Effective component-oriented software development relies on the ability to identify and retrieve adequate components and on the feasibility to distribute components with all necessary information. We discuss the principal component management activities in the next section, followed by a presentation of the FRAMBOISE component schema that determines the information to be registered with a software component.

Component Management Activities

Component management principally involves the storage of components as well as classifying and packaging them for subsequent distribution.

Storage Components are preferably stocked in component repositories that contain the software components with all relevant information about them. Repositories are sometimes subdivided into repositories providing general purpose components (*local repositories*), *domain-specific repositories* containing special-purpose components and so called *reference repositories* [Moo94] that accumulate meta information to find components in other repositories.

Cataloging Components must bee cataloged in order to be able to identify and retrieve them subsequently in an efficient way. The catalogue information should consist basically of precise specifications of what components do and what platform requirements they have. Research has concentrated on how to catalog and retrieve components but there are no methods that are proven to work with components of substantial complexity. There are various indexing vocabularies to classify components, beginning with generally applicable approaches like free text, keyword classification and enumerated classifications (alike ISBN numbers) to more specialized classification schemas that rely on specific facets or attributes (for a comprehensive overview cf. [Sam97]).

Packaging Components must be packaged in such a way that they are deliverable and can be administered at their deployment site. A reasonable container is necessary if, for instance, a component consists of various modules or objects and exhibits various interfaces. Moreover, a component package typically includes meta information about the components as well as additional tools for deployment. Finally, component packages often are certified by message digests in order to ensure the customer that the respective package can be trusted.

4.1.5 The FRAMBOISE Component Model

From a component-oriented point of view, the construction system devised in this thesis represents a component framework, i.e, a second-order framework as described in Section 4.1.3. This section establishes the characteristics of software components applied in FRAMBOISE, as well as the paradigm according to which they are connected. Finally, the kind of information to be stored and packaged together with a component is specified.

Software Components in FRAMBOISE

Components in FRAMBOISE have the following characteristics:

- The rigorous conception of a software component given in [Szy97] (cf. Sec 4.1.1) is applied in FRAMBOISE. Thus it is avoided that one looses the focus by subsuming a variety of well-defined software constructs like templates, macros, objects or modules under the term software component.

- Components shall represent significant functional units of composition of the future ECA Systems. Their *granularity*, however, may vary such that components can be *coarse-grained* by providing high level functions like complex event detection or they may be *fine-grained* to provide low level tasks such as the mapping of system specific parameters to an ECAS internal representation. However, we rule microscopic components out – "Component assortments that offer a hundred of different implementations of stacks and queues are not what the component market is waiting for [Szy97]".

- Component interfaces in FRAMBOISE

  – define the operations that clients can invoke on the respective component,

  – include interface definitions that a component requires to be successfully deployed, and

  – are contractually specified by means of Hoare triples.

  The more sophisticated approaches to specify interfaces presented in Section 4.1.1 are not practical for FRAMBOISE, because interfaces are not only a means

to describe the external view of a component but they are also necessary to enforce subsequently the integrity of a composite system. Thus we adopt a conception of component interfaces which is applicable for current implementation techniques.

Note that even this simple perception of a component interface is a conceptual view that might be compromised by a chosen implementation technique. For instance Eiffel [Mey92b] is to our very best knowledge the only one among the more popular programming languages that enforces Hoare triples. In other languages one is required to insert corresponding assertions manually in the respective code. However, interface definitions obeying the above requirements can be mapped rather straightforwardly to the most up-to date programming systems.

- Components in FRAMBOISE are able to reify. Since FRAMBOISE is a research project, the risk of component decompilation – the chief drawback of reification – is neglectable.

- FRAMBOISE relies on whitebox abstractions of components in order to support an ADBI (cf. Sec. 3.4.1) to implement new components. A subsequent transfer to glassbox or even blackbox abstraction is out of the scope of this thesis.

- In order to prevent a future proliferation of rather similar components, the reimplementation of components that nearly match the requirements of a specific application should be avoided. Thus, components shall exhibit features to be adjusted with a well defined impact (e.g., so-called *properties*) in order to give ADBIs means to adapt preexisting components for their means without tampering at the actual component implementation.

Note that components are regarded in this thesis rather as *abstract components* in a way that they could be provided for arbitrary component infrastructures. Thus the FRAMBOISE component model incorporates no proposal of a component infrastructure. The prototypical implementations, however, are performed by means of the Java programming system whereas component prototypes are realized as JavaBeans™ [Javb] (cf Chap. 7).

Component Connection in FRAMBOISE

Components are assembled by plugging them into the appropriate insertion point of the glue provided by the component framework. These insertion points are provided in accordance with the specifics of the respective ECAS. A scripting facility enables the ADBI (and the generator tools addressed in Section 3.5.1) to configure the glue correspondingly.

The implicit invocation of component services as they are provided by message-oriented connections support reuse and ease system evolution [SN92]; hence these

mechanisms are mandatory for FRAMBOISE. The flip side of the coin is that components basically relinquish control over the computation performed by the overall system if they interact by a means of implicit invocation. A component that sends a message implicitly neither can rely on the order the components interested in this service are invoked nor does it know when all invocations associated with a specific message are finished. Furthermore, reasoning about correctness can be problematic [GS94], in contrast to traditional invocation mechanisms like procedure calls, where one needs only consider a procedure's pre- and postcondition when reasoning about an invocation of it.

It is, however, mandatory that the construction process of FRAMBOISE asserts to an ADBI that the rule execution behaves in accordance with the rule execution semantics defined by the knowledge and rule execution model of the respective ECAS. As a consequence, message-oriented connections shall not be discernible for an ADBI but they shall be marshaled by facilities of the component framework implementation in order to ensure correct behaviour. From the point of view of an ADBI a service that is declared in the "required interface" (cf. Section 4.1.1) shall be invoked explicitly. Thus it must be feasible to circumvent the message-oriented connections by invoking a service of a known (i.e, whose availability is ensured by the component framework) component directly.

The FRAMBOISE Component Schema

A component model also comprises the specification of the information to be stored and packaged together with a software component. This specification is called the *FRAMBOISE component schema*. Inspired by a presentation of the REBOOT Component Model [Kar95] the FRAMBOISE component schema is presented in an entity relationship notation as depicted in Figure 4.1 Note that the granularity of the information related to the software component is not fixed. It is conceivable to subdivide any information – represented by the shaded boxes – related to the component into smaller entities (e.g., test support might be decomposed into single test cases) with independent relations to the reusable component. For the sake of readability, however, these entities are not decomposed any further here.

Classification Information The *classification* of a component is the information intended to identify and retrieve a component. The chosen classification schema is discussed in Section 8.2 in the context of the actual component provision.

Quality Information This information describes the quality and reusability of a component in order to enable an ADBI to decide whether a candidate component meets the respective requirements. Furthermore, the qualification information contains the reuse history of a component. A reuser can record any critical or positive comments about the component, any problems that occurred when reusing it and how the problems where solved.

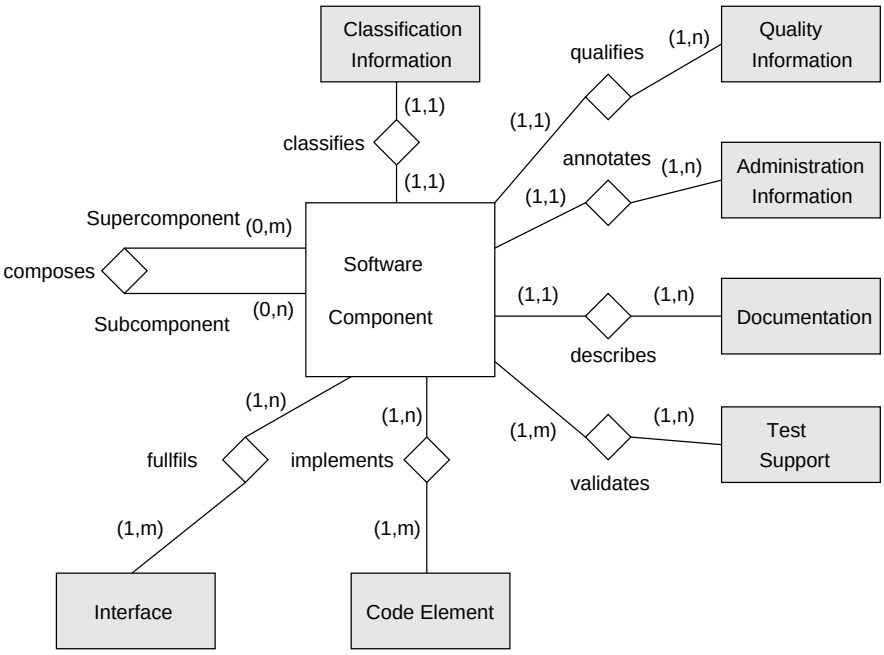

Figure 4.1: The FRAMBOISE Component Schema

Administration Information It includes general information like references to the developer and the current maintainer and a development as well as a maintenance history. Furthermore the administration information includes attributes pertaining to an authorization schema so that the administrator of the component repository is able to grant rights to developers for sensitive components in order to enforce, for example, glass box reuse for a sensitive component. In a commercial environment pricing information would also be attributed to the administration information, however, for the FRAMBOISE research project, we leave this element apart.

Documentation The documentation complements the classification and the quality information by describing the exact functionality of a component and how this functionality is internally achieved (e.g., by explaining internal data structures of a component). Hence the documentation assists an ADBI by selecting the appropriate component out of a set of similar ones as well as by adapting a component. In a nutshell, the documentation corresponds to a description of building blocks in a class or module library.

Test Support Although components are tested before inserting them into the repository, it is necessary to test them again when they are reused, maintained or adapted. Since the construction of good test suites is a costly activity, it is mandatory to have these test suites available together with the reusable component. Test support compre-

hends test fixtures (programs written specifically to test the component), test data on which to run them and information about the expected results. Note that the relationship between a software component and the test support is N:M, because a reusable component may have many alternative test suites whereas one test suite may be applicable for various components.

Interface In the FRAMBOISE component model, interfaces are considered in the sense of Section 4.1.1 as a contractual specification of the services to be fulfilled by a component. Interfaces are stocked as distinct entities in the FRAMBOISE component repository, which are related with a M:N relationship to the components that fulfill the respective interfaces.

Interfaces are preferably defined by means of specialized *interface definition languages* (IDLs) which abstract from specific implementation languages. The interface definitions are subsequently processed by so-called IDL-compilers that map among other things the interface definitions to constructs of specific programming languages and register the components interface in repositories. Nowadays most prominent examples are OMGs CORBA IDL and Microsoft's COM IDL. An approach proposed in [BBK+97] is adopted. It provides a convenient way to make the context dependencies of a component explicit. This approach is presented in greater detail in Section 7.2.1.

Code Elements Program constructs like modules or classes and their compiled binaries do not comply with the strict notion of a software component given in Section 4.1.1. Thus the code elements (i.e., source and binary code) that are required to implement an atomic component (i.e., those not composed out of other components) are stored as separate entities in the component repository.

Composite Components Composite components are at least partially composed out of other components which are called *Subcomponents*. In order to enable an ADBI to navigate from *Supercomponent* to candidate subcomponents and vice versa, the composite relationship is also registered in the component repository. A component does not mandatorily participate in this relationship, i.e, an atomic component is never a supercomponent whereas there are top level components (e.g., entire ECASs) which are never subcomponents in the context of FRAMBOISE.

4.2 The Architecture Model

Architecture models are means to express the architectural level of system design, which is concerned with the gross structure of a system as a composition of interacting parts [MKMG97]. Thus, architecture models are a collection of conceptual tools for

describing the structures of software systems[1], functional aspects of their constituents, the relationships between them as well as consistency constraints.

Even though it has long been recognized that the appropriate software architecture of a system is a key element for its long-term success, architectural design is still a relatively young software engineering discipline. Therefore, a well-accepted taxonomy of architectural paradigms did not appear yet, let alone a fully-developed theory of software architecture [GS94]. Strictly speaking, there is not even a single, universally accepted definition of the term software architecture. Instead, a plethora of definitions have emerged that typically agree on the major ingredients of an architecture – structure, components and connections among them – but vary widely in their details and are not interchangeable. It is therefore mandatory that we establish an architecture model by defining how the various terms are perceived in FRAMBOISE.

The subsequent paragraphs are chiefly a compilation of the literature into a coherent ensemble that enables a systematic architecture design. Upon clarifying the notion of a software architecture in Section 4.2.1, the concept of *architecture styles* that encompass the key elements of architecture modeling is introduced. Subsequently the terms *reference model* and *reference architecture* are clarified. They are means to design architectures that are applicable to design families of software systems. Section 4.2.4 presents an *architecture definition language* that enables the formal description of software architectures. Finally, the architecture model underlying the FRAMBOISE component framework is introduced in Section 4.2.5.

4.2.1 The Notion of a Software Architecture

In this thesis architectures are conceived according to the following definition.

> The software architecture of a program or computing system represents the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them [BCK98].

The statement that an architecture may have multiple structures means that different aspects (e.g., the module or class structure, the process structure, data flow etc.) of the *same* architecture may be highlighted by individual structures. The term *externally visible* properties refers to those assumptions that other components can make about a specific component, such as services provided, performance characteristics, shared resource usage etc. Behavioral aspects of components may be part of the architecture insofar as that behavior affects other components. Due to this emphasis of the externally visible component properties, the above definition of software architecture harmonizes particularly well with the requirements of component-based software engineering.

[1] Unless stated explicitly otherwise, we apply the term “architecture” always restricted to the architecture of software systems

Architectures are also considered as a specific reuse technique [Kru92], representing significant design decisions that are reused as a whole. The reuse of architectural and design experience is regarded for such a beneficial impact on development costs and software quality of systems with similar requirements, that [Szy97] even claims it as the "probably single most valuable strategy in the basket of reuse ideas".

4.2.2 Architecture Styles

In current practice, the codification and reuse of architectural designs has occurred chiefly through the informal use of architectural idioms. For example a system might be defined architecturally as a "client-server system". Such classes of architectural idioms have been termed as *architectural styles* [AAG93]. An architectural style characterizes a family of architectures that are related by shared structural and semantic properties. It consists of a vocabulary of design elements, patterns of their runtime control and data transfer and finally a set of design rules that determine the composition of the design elements.

Architecture design research has identified a variety of architecture styles (for a comprehensive overview cf. [Sha96, MKMG97]) and classified them as *architectural patterns* that help a designer to map these styles[2] to the needs of the problem at hand in order to determine the basic scheme of an architecture.

The description of each pattern can be structured as follows [Sha96]

Problem What problem the pattern addresses i.e., the characteristics of the application requirements that lead a designer to select this architecture style.

Context Which aspects of the computation environment or implementation constraints restrict the designer in the use of the respective style.

Solution The system model captured by the architecture style. It includes a description of the *components* that are part of this style, the *connectors* that mediate interactions among components and the *control structure* that governs the execution.

Diagram A figure showing a typical pattern, annotated to show the components and connectors.

Significant Variants Major variants of the basic style (if any).

Examples References to examples or more extensive overviews of systems that apply this architecture style.

This roster enables the classification of architectural styles as shown in the following example.

[2]Note that architecture styles are rarely applied in their pure form. Most systems involve some combination of several styles.

Architecture Style "Layered Systems"

Probably the most prominent architecture style is represented by the so-called *layered systems*, where components are assigned to layers in order to control component interaction.

Problem This style is adequate for systems that involve distinct classes of services that can be arranged hierarchically. Typically there are basic system-level services, covered by layers that provide utilities used by many applications and on top layers for application specific tasks. There are several benefits of this architecture style that have been empirically verified [ZWH95]. First, layered systems support design based on increasing levels of abstraction, thus enabling implementors to partition a complex problem into sequences of incremental steps. Second they support enhancement because changes to the functionality of one layer affect at most the two neighboring layers. Third, they support reuse because different implementations can be used interchangeably, provided that they support the same interfaces to their adjacent layers.

Context Layers are often used at higher levels of design, using different styles to refine the layers.

Solution

- *System Model:* Strict hierarchy of opaque layers, i.e., each level communicates only with its direct neighbors.
- *Components:* Usually composites which are most often collections of procedures.
- *Connectors:* Typically procedure calls.
- *Control Structure:* A single thread.

Diagram Shown in Figure 4.2. Layered architectures are sometimes drawn as concentric circles. In that case, *lowest* layers become *innermost*.

Significant Variants Not all systems are easily structured in a layered fashion because it can be quite difficult to find the right levels of abstraction. Even if it is feasible to structure a system *logically* into layers, performance considerations may require closer coupling between high level functions and their lower level implementations. In fact, layered systems are in practice frequently not "pure" because functions in one layer may call operations that are located in other layers than the one immediately below. This is called *layer bridging*.

Examples One of the first concise presentation of this idea, in the context of the operating systems architecture, was Dijkstra's article on the THE operating system [Dij68]. The architecture of DBMSs is often also organized into layers [Hae87, HR99].

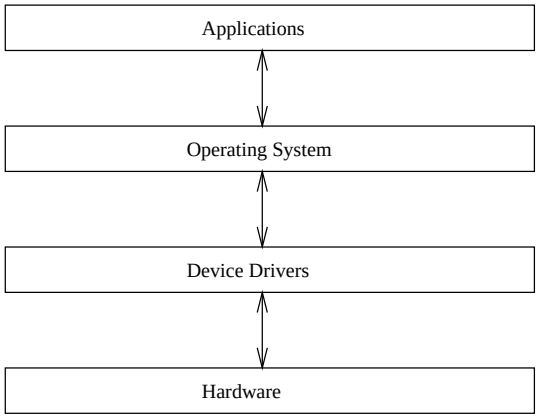

Figure 4.2: An Example for the Layered Architecture Style

The use of architecture styles has a number of significant benefits [MKMG97, PW92]:

- Architecture styles leverage design reuse, because it enables to reapply routine solutions with well-understood properties with confidence to new problems.
- Application of architecture styles also can lead to significant code reuse, because the invariant aspects of an architectural style often lend themselves to shared implementations.
- It is easier for others to understand a system's organization if conventionalized structures are used.
- Standardized architecture styles support interoperability.
- By constraining a design space, an architecture style often permits specialized, style-specific analysis (e.g., deadlock detection for client-server message passing).
- It is usually possible and desirable to provide style-specific graphical descriptions of design elements. This makes it possible to provide graphical renderings of designs that match the engineers' domain specific intuitions about how their designs should be visualized.
- Architecture style embodies those decisions that typically suffer erosion and drifts (also known as *architectural decay*) during implementation and subsequent evolution when a software system undergoes maintenance.

Architecture styles are less constrained and less complete than specific architectures. There is, however, no hard dividing line between where architectural style ends and software architecture begins [PW92].

4.2.3 Reference Models and Reference Architectures

After a critical mass of systems for a specific purpose have been developed, so-called *reference models* are formed. These are commonly accepted functional decompositions of the respective task into parts that cooperatively solve the respective problem. It is for instance possible to enumerate the standard elements of compilers (e.g., lexical analyzer, code generator etc.). However, reference models are no architectures in the sense of the above definition, because the separation and interaction between the components is not specified precisely and can vary from system to system.

In contrast to a reference model that divides the functionality of a system, a *reference architecture* is the projection of that functionality onto an effective system decomposition, i.e., onto components that provide the functionality defined in the reference model and the interactions between the components. The mapping between elements of the reference model and components of the reference architecture is not necessarily one to one (even though this may occur occasionally). Thus a software component may implement a part of a function or it may provide several functions of the reference model.

Reference architectures are the way architectures are reused across multiple systems. They are complementary to architecture styles, because the latter involve one single "architecture philosophy" applicable for arbitrary application domains, whereas a reference architecture may incorporate several architecture styles but is outlined for one specific purpose. Typically, a reference architecture is developed by applying one or several architecture styles on a reference model to achieve the final system decomposition. The dependencies between reference models, architecture styles and implemented software architectures are depicted in Figure 4.3.

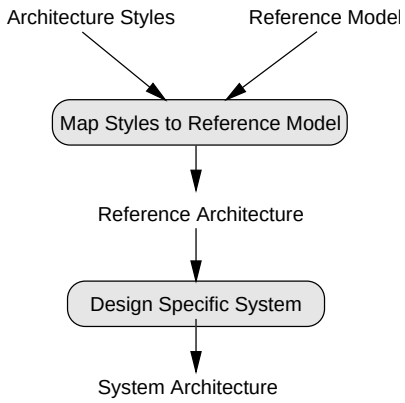

Figure 4.3: Stages of Architecture Design

The benefits of reference architectures are threefold: First, they enable the application of generative construction techniques which require that the final products are aligned according to a reference architecture [BCK98]. Second, they are a basis for

designers to derive the software architecture of the respective system under development by refining and extending the reference architecture. Third, if the target system is developed by composing it of prefabricated software components, reference architectures help a designer to identify constraints on the components that components must fulfill to be applicable. Thus, by studying a reference architecture, a designer is able to understand the concepts embodied in a component library which in turn has an enormous leverage on the ease to find adequate components. Note that the opportunities to reuse components generally improve when specifications for components are constrained at least at the architectural level [PW92].

4.2.4 The Architecture Specification Language WRIGHT

Informal descriptions of software architecture give an intuitive picture of the respective system's structure but do not provide enough aspects to begin a detailed component design. Representing an architecture largely by annotated box-and-line drawings makes it hard to pin down specific effects of service invocations and component interactions. For instance, this kind of abstraction is not concise enough to resolve directly how the various elements in an ECAS and its associated DBMS are synchronized in order to process rules triggered by database events.

The so-called *architecture description languages* (ADLs) are a linguistic approach to describe software architectures formally and address the shortcomings of informal representations, allowing early analysis and feasibility testing of architectural design decisions. It is nowadays accepted that ADLs are a sensible approach to specify software architectures authoritatively, even though nearly all of these languages (for a comprehensive overview cf. [Med96]) are still in the research stage and it is premature to choose any of them as most promising.

The architecture description language WRIGHT [AG94, AG97, All97] is used to specify software architectures in 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 behaviour and coordination of components and connectors are specified in WRIGHT by means of a notation based on the process algebra CSP ("Communicating Sequential Processes") [Hoa85]. This formalism 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[3] and communicate with each other through well-defined channels. Besides being a notation, CSP is also a collection of mathematical models and reasoning methods to analyze *concurrent systems*. Thus, WRIGHT enables a designer to write down an architectural description precisely and to validate this description subsequently for con-

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

sistency and completeness. A detailed presentation of WRIGHT is given in Appendix A.

4.2.5 The Architecture Model of FRAMBOISE

This section concludes the above discussions by establishing the architecture model of FRAMBOISE.

Purpose The architecture model of FRAMBOISE focuses on the structure of ECA Systems. This structure comprises software components, the externally visible properties of those components and the relationships among them [BCK98]. The focus of the architecture model is the specification and analysis of interactions between architectural components.

As a construction system, FRAMBOISE is outlined to furnish a family of similar products, namely the various ECA Systems. Thus the architecture model is outlined to specify reference architectures as discussed in Section 4.2.3 instead of system architectures. The reference architecture of the ECA Systems serves as a blueprint for the ADBIs to compose a specific ECAS.

Ingredients The architecture model of FRAMBOISE is formed by the following ingredients:

- *Components* represent the primary computational elements and data stores of system. Intuitively they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as client, servers, filters, objects blackboards and databases.
- *Connectors* represent interactions among components. Computationally speaking, commectors mediate the communication and coordination acitvity among componants. That is, they provide the “glue” for architectural designs and intuitively, they correspond to the lines in box-and-line drawings.
- *Architecture Patterns* that generalize solutions for specific architectural (cf. Sec. 4.2.2).
- The *architecture definition language* WRIGHT to formalize the respective architecture.

Procedure The reference architecture of the ECA systems is devised according to the procedure depicted in Figure 4.3, i.e., a reference model of ADBMSs is mapped onto a set of architecture styles. Architecture design is accomplished if the respective architecture is formally specified in WRIGHT. Thereby the respective design must hold against the correctness criteria of this architecture description language (cf. App. A).

Chapter 5

Process Definitions

In order to furnish an active database construction system it is necessary to define the so-called *instance unbundling process* and the *instance rebundling process*[1] (cf. Sec. 3.5.2). These two processes guide the activities to decompose ADBMSs into reusable components (unbundling) and and to rebundle them into novel systems.

Since bundling-oriented database construction is still in its infancy, there are no proper process models to perform un- and rebundling. Hence this chapter develops a novel unbundling and a (re-) bundling process from scratch. The chapter is organized as follows. Section 5.1 discusses principal issues concerning software development *for* reuse and software development *with* reuse and how these activities are combined. Subsequently the FRAMBOISE unbundling process is presented in Section 5.2 and the rebundling process in Section 5.3. Section 5.4 concludes the chapter.

5.1 Reuse-Oriented Software Processes

In traditional software process models[2] such as the waterfall model, software reuse is not considered to be an explicit part of the process. Instead, these life cycle models are outlined to develop systems from scratch [Sam97] (even though some reuse might happen in the implementation when programmers adapt previously written code) with the primary goal to get the respective system finished. Typically no attention is paid to whether there might be some components of the system to be considered for reuse in other projects.

Considering reuse implies that the overall software development process is split into two complementary (Sub-) processes: namely *developing software for reuse* which is denoted with the term *component engineering*[3] when reusable components are pro-

[1] For short this processes are addressed simply as unbundling and rebundling process respectively.

[2] A software process model is defined as a set of activities, methods, practices and transformations that people use to develop and maintain software and the associated products [SW94]

[3] Since FRAMBOISE is a component framework, to software development for reuse is referred to exclusively as component engineering.

vided [Sam97] and *application engineering* [Sam97] which is concerned with the development of systems by reusing components. This division is sketched in the so-called *twin life cycle model* [Kar95] (cf. Fig. 5.1). The artifacts provided from the

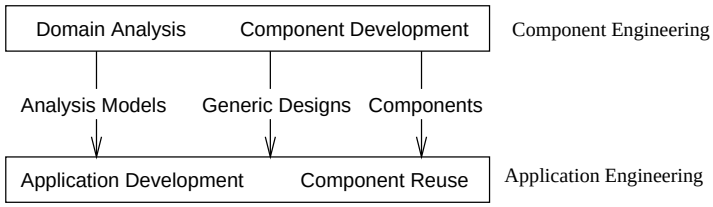

Figure 5.1: The Twin Life Cycle

component engineering process are usually stocked in repositories and are retrieved therefrom when application engineering takes place. Hence repositories form the tangible link between these two activities [Sam97]. The principal issues of the two development processes are discussed in the subsequent paragraphs.

5.1.1 Component Engineering

Component engineering (respectively development for reuse) is defined as the *planned activity of constructing a software component for reuse in other contexts than for which it was initially intended* [Kar95]. It is a widespread misconception that a component library can be created cheaply by extracting and documenting components of existing monolithic systems. Software building blocks that are created as part of a specific application are unlikely to be reusable immediately [Som92], because they are typically streamlined towards the requirements of the system in which they are originally included. It is therefore necessary to decompose the respective original systems to reengineer their constituents so that they become reusable. This form of component engineering is referred to as unbundling (cf. Sec. 3.1.3).

In spite of gaining the components by unbundling preexisting systems or by developing them from scratch, the essence of component engineering remains the same. In either case component engineers must identify the requirements of potential users with similar needs and design a general solution which can be adapted economically to satisfy as many requirements as possible [BB91]. We discuss in the next section the principal activities of component engineering, followed by a presentation of component generalization techniques.

Principal Activities

Component engineering basically involves all activities typically found in traditional software development processes (e.g., Analysis, Architecture Design, Implementation

Testing etc.). The principal difference is that the functionality of the identified components is extended and generalized in order to capture the requirements of future applications.

The following steps are considered as necessary to develop software for reuse [Kar95]:

1. *Collecting the set of requirements to make an initial solution.*

2. *Definition of an initial solution or identification of previous solutions to the same set of requirements.* A cost estimation for developing the actual solution should be made or updated at this time.

3. *Identification of possible generalizations*. This is the inventive step in which the component engineers try to see the generality in the initial requirements and solution. At this step the components reuse potential should be discussed with the product management and the plan for the subsequent steps should be redefined.

4. *Identification of potential reusers and their requirements.* Information about potential reusers is important to ensure that the generality to be included in the components is really justified.

5. *Estimation of the cost and benefit of added functionality.* For each added requirement the component engineers should estimate the benefit of the reusers who will reuse it as well as the extra cost (e.g., time to understand and rule out) for reusers who do not need the respective feature. Furthermore, one should also estimate the effort a reuser needs to develop the functionality from scratch, and the probability that the component will be reused.

6. *Analysis of the added requirements with respect to invariants and variation.*

7. *Proposal of a generalized solution with specifications and cost estimates.*

8. *Presentation of the solution to reusers and reuse experts for validation and approval.* Based on this input, it is decided whether the component is developed for reuse.

9. *Development and documentation of the solution.*

In the present form, these steps are generic and are basically applicable at any stage of a development process.

A typical project to provide a component framework basically proceeds as follows: Starting with a domain analysis, which collects among other things a number of examples, a first version of the component framework is designed. This initial version typically relies almost exclusively on whitebox abstractions and enables just the implementation of these examples. Afterwards the component framework is used to build

applications which point out critical points in the design, i.e., the parts of the framework that are hard to change. Experience leads to improvements in the framework, which permit more effective blackbox reuse. Eventually the component generalizations are good enough that suggestions for improvement become rare. At some point the developers have to decide that the component framework is finished and release it.

Component Generalization

Components are basically generalized by means of four techniques [Kar95] which are applicable for components of all sizes:

Widening A component is generalized by widening its scope, i.e., its requirements are extended insofar as it is feasible to identify a set of requirements that are not contradictory.

The benefit of widening is that a widened component can be reused without further modification and that it is more likely that it is reused in future applications. Disadvantages are, that the initial development costs of such a component are higher and that the component may become unnecessarily complex. Moreover, the component may have more functionality than is used in most reuse contexts which may lead to inefficiencies in execution speed and memory usage.

Narrowing This is the complementary operation to widening as one narrows the scope of components and limits their functionality to a set which is required by several applications. This is achieved by identifying functionality that is common to the known application cases and representing it by an abstract component. Narrow components build the base for various extensions that are implemented in *separate* components.

Advantages and drawbacks of narrowing are complementary to those of widening. A specific disadvantage of narrowing is that an immense number of similar components may make it difficult to choose the right one.

Isolation Specific requirements are isolated to certain components or parts of components. Thus it is feasible to construct the other components rather independently of whatever specialization is chosen. Isolation is typically used to separate components from system-specific parts like operating systems or hardware.

Configurability Configurability means that we build a set of smaller components which can be composed in various ways in order to meet different needs instead of building a (large) component satisfying all requirements. This approach is especially useful for optional requirements and for the separation of variant and invariant functionality.

Generalized components may be reused even if the requirements of a certain software system are more specialized and limited than the broad and general functionality offered by the candidate component. The eventual overhead caused by unused functions

must be seen as the price for increased productivity [Sam97]. Redundancy is correspondingly taken into account for the sake of higher productivity, i.e., it may occur that different components coincide in some of their functionality but all components should be incorporated in the final system because of some other functionality.

Even though generalization is imperative to achieve reusable components there is also the danger of over-generalization so that a proper balance between reuse potential and ease of implementation must be found. However, it is far from trivial to decide whether a component is reusable or not. [JF88] even concludes that the only way to find out if software is reusable is to reuse it. It is therefore apparent that component engineering bears more uncertainty and more overall project risks than the development of a standard application. Not surprisingly the principal activities to develop software for reuse as they have been presented in the previous section, include various controlling activities such as the evaluation of possible generalizations or the verification of their reuse potential etc.

5.1.2 Application Engineering

Application engineering based on reusable components typically proceeds as shown in Figure 5.2. This procedure implies a rather opportunist form of software reuse,

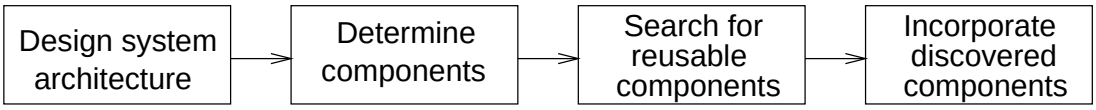

Figure 5.2: Software Development with Reuse

because the availability of reusable components is not taken into account during the system design phase. However, one can observe that other engineering disciplines typically base their system designs on preexisting components [Som92]. That means that the system requirements are modified according to available components as far as the resulting set of requirements ensures an adequate solution. As a consequence one must accept requirements compromises and the design might be less efficient. However, in well-established engineering disciplines, the lower costs of development and increased system reliability usually compensates for this.

A corresponding a *reuse-driven* software development process is depicted in Figure 5.3. Note that the most expensive task here is the understanding of the retrieved components. The cost of understanding existing software at all levels, not only at the implementation level, is usually seriously underestimated (e.g., [Kar95] assesses the expenses for maintenance at 40 to 70 percent of the total cost).

The schema depicted in figure 5.3 addresses only the initial activities of application engineering. It results in the specification of the components required to provide the final system. More precisely this specification

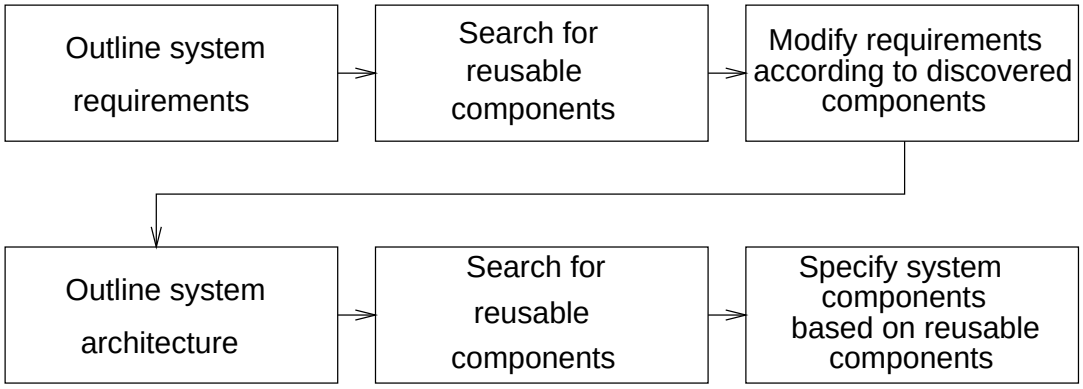

Figure 5.3: Reuse-Driven Development [Som92]

- identifies the components that can be incorporated "as is" in the future system,
- names the components that must be modified or adapted[4] and specifies these modifications, and
- specifies component that must be developed from scratch.

Once the system components are specified, they have to be provided either by retrieving them from the repository, or by modifying/adapting existing ones or finally by implementing them from scratch. Afterwards the components have to be integrated into a coherent system. Finally, there are usually "debriefing" activities such as the incorporation of new components into the repository as well as the evaluation of components for their reuse potential and reporting of reuse experiences (cf. [Kan87, HC91, Kar95]).

In order to enable systematic reuse, all these activities must be incorporated in a software life cycle. A strict sequential procedure as suggested in the above figures is, however, not practical for a component-based application engineering. Such procedures basically rely on static requirements and are not suited to deal with incomplete and inconsistent specifications as they occur in reuse-driven development processes.

Integrating the reuse activities into the well-known spiral model [Boe88] is suggested in [Sam97]. Basically the tasks related to reuse are attributed to the appropriate quadrant of the spiral (cf. Fig. 5.4).

- *Quadrant I:* The first quadrant of the spiral involves determining the objectives of the system to be developed, identifying alternative solutions and constraints imposed on them. Concerning reusable components, this means that application

[4]In the literature the terms *modification* and *adaptation* are not clearly differentiated. The perception of [Sam97] that suggests using adaptation for minor changes that were in some way planned by component developers (e.g., parameterization) is adopted in this thesis.

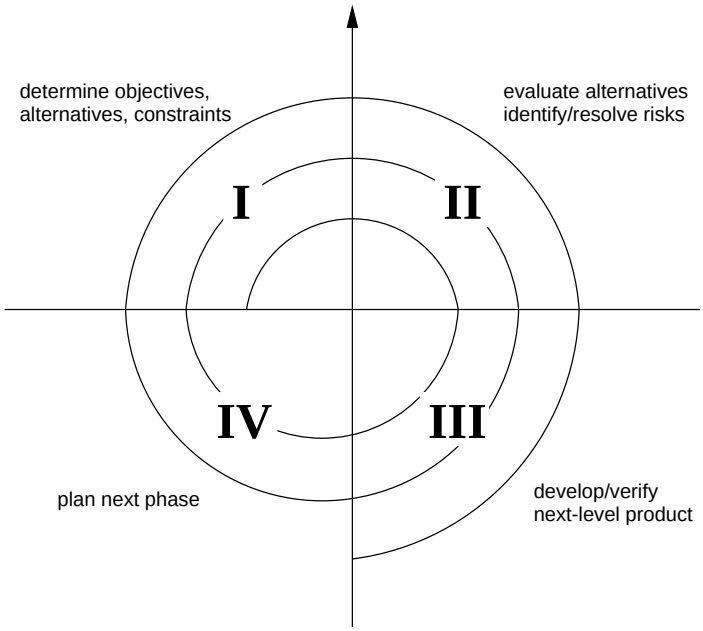

Figure 5.4: Boehm's Spiral Model

engineers must understand the basic problem and identify components that could be used to compose the target system.

Result of quadrant I is a number of candidate components and various solution approaches depending on the functionality of these components.

- *Quadrant II:* In the second quadrant one evaluates alternatives and identifies and resolves risks. Thus, the application engineers determine the need to modify or adapt components, the effort necessary to accomplish the modifications and the risk involved in doing so. Different solution approaches elaborated in the previous quadrant are evaluated by risk and effort assessment. Risks have to be determined by gathering more information about components or by experimenting with them.

  This quadrant results in the decision on the solution to be pursued and which components to reuse, respectively what modifications/adaptations to make.

- *Quadrant III:* The third quadrant of the spiral comprises the development of the next-level product. That means that the selected components have to be modified/adapted and integrated into the subject system.

  The result of this quadrant is the implementation of the system part to be developed.

- *Quadrant IV:* The fourth quadrant of the spiral involves planning for the next phases. Concerning reusable components, the application engineers look back

and evaluate reusability prospects of modified or new components in order to incorporate them into the component repository. Furthermore, they add feedback about reused components and experiences to the corresponding repository entries.

The use of the spiral model provides the means to test the development options for an application in a limited context with controlled risks as it is necessary to experiment with reusable software components. The spiral model is therefore particularly suited as a basis for reuse-driven software development as it is suggested in Figure 5.3.

5.2 The FRAMBOISE Unbundling Process

The instance unbundling process of FRAMBOISE defines the procedure according to which components are engineered. Thereby one must take into account that the intrinsic uncertainty and risk of component engineering projects is aggravated in FRAMBOISE for the following reasons:

- From the software engineering point of view, ADBMSs are still an immature domain. It is therefore probable that certain mistakes in the domain analysis are only discovered when components are designed or an ECAS is built.
- The component framework makes the parts of the design explicit that are likely to change. Getting these interfaces and shared invariants right is a hard task. Basically the only way to learn what should be changed and how it shall be represented is by experience. Thus it is foreseeable that component designs will be revised in later phases of unbundling.
- Paper designs are necessary abstractions to design a component framework, however, they are not detailed enough to achieve the right component generalizations. Instead, appropriate component generalization depends on proper abstractions of examples found during domain analysis. In that sense the rarity of industrial-strength active database applications increases the risk that the component framework later undergoes major changes due to inadequate generalizations.

It is therefore of paramount importance to base the instance unbundling process on a method that is able to cope with frequent modifications and iterations in the development process. A method called *Extreme Programming* is a viable procedure to perform unbundling. This technique is discussed in the next section, followed by the presentation of the the actual FRAMBOISE unbundling process (Sec. 5.2.2).

5.2.1 Extreme Programming

Extreme Programming (XP) [Bec99, Bec00] is a lightweight software development method conceived to address specific needs of software development conducted by

small teams in the face of vague and changing requirements. XP represents in its essence a novel form of team-oriented, evolutionary software development. Unlike conventional methods in which planning, analyzing and design activities are rather outlined to cover the overall project, XP starts with a quick analysis of the entire system whereby analysis and design decisions are continuously made throughout development. A system under development is put into production in a few months, before solving the whole problem. Nevertheless XP is by no means a chaotic "code and fix development technique" but relies on a number of practices which are strictly enforced to enable a fine-tuned control of the development process and to foster the communication among developers and customers.

The XP Development Cycle The XP development process is a cyclic activity that is performed as follows: In the analysis phase so-called *stories* are defined. Stories can be thought as the amount of use case that will fit on an index card. Each story must be business-oriented, testable and it must be feasible to estimate the effort to implement it. Otherwise the definition of a story is not considered as completed and it might eventually be necessary to split a story into smaller ones or to experiment with vertical prototypes. The effort to furnish the software for a specific story is typically measured in so-called *ideal engineering time*, which is defined as the time without interruption where developers can concentrate on their work feel and fully productive.

The stories are grouped into *releases* which settle the stages according to which the project proceeds. Releases are realized by means of several *iterations* whose aim is to put into production some new stories that are tested and ready to go. New releases are made often – anywhere from daily to monthly.

The process starts with a plan that defines the stories to be implemented and settles how the team will accomplish it. *Iteration planning* starts by asking the customer to pick the most valuable stories, among those to be implemented in this release. The team decomposes the stories into *tasks*, i.e., units of implementation that one person could implement in a few days.

Task implementation starts with implementing the test cases in order to perform unit testing. Programmers write (unlike to the cleanroom approach [CM90]) their own tests whereby they write it before they code. Thus programmers must first of all establish a list of test cases which is continuously condensed. The tests are collected and they must all run correctly. New code is integrated with the current system within small time intervals (usually a few hours). During integration, the system is built from scratch and all tests must pass or the changes are discarded. The design of the system is continuously *refactored*, i.e., it is evolved through transformations of the existing design that keeps all tests running.

Team Organization XP emphasizes close cooperation among the team members. Probably the most striking feature of XP is the so-called *pair-programming*, i.e., all production code is written by two people at one screen/keyboard/mouse. The team

works in a large room with small cubicles around the periphery whereby pair programmers work on computers set up in the center of this room.

A customer joins the team full-time and is responsible for the release planning and the provision of larger-grained functional tests. Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by stories in the respective iteration. The shape of the system is defined by a *metaphor* or a set of metaphors shared between the customer and the programmer.

Finally the production code is considered as collectively owned by the team members. That means that every programmer is allowed to improve any code anywhere in the system at any time.

5.2.2 The Actual Unbundling Process

In order to establish the FRAMBOISE unbundling process, the XP development cycle must be tailored to the main activities of component engineering described in Section 5.1.1.

The practices of XP are obviously appropriate to generalize components, because component generalization requires continuous refactoring in order to achieve better generalizations. Extreme programming has originally been devised for the development of applications. We suggest the following modifications in order to apply XP for the provision of reusable software components:

- Unbundling ADBMSs includes the development of several applications that use specific ECASs in order to provide the feedback that component developers need to refactor the generalizations. Of course these “pilot” applications will not benefit from reuse – the benefits usually do not start to show until the third or fourth application [FSJ99]. In contrast, their development is probably more expensive on account of additional iterations which are due to component refactoring. Since these extra costs are caused by unbundling activities they must be compensated from the funds of the profiting unbundling project.

- FRAMBOISE is developed for third-party reuse. Thus the role of on-site customer also includes programmers providing the applications that use the example ECAS.

- Most of the stories applied in the unbundling process of FRAMBOISE are typically not business cases as it is common for XP. Instead the stories represent specific facets of active database behaviour as they are outlined in table 3.1 or they stem from ECA rules specified for the “pilot” applications. For instance a story might be ”enable a conflict resolution scheme based on relative priorities” or process the rule

```
ON TIMES(10,UPDATE TO employee.contracts):SAME customer
```

```
IF salary < 5000
DO increment(employee.bonus)
```

- The reference architecture of ECASs captures the system metaphor so that everyone in the team knows how the system works. The architecture model introduced in Section 4.2 is a means to describe and analyze the system metaphor to set a basic map for the future component provision.

The FRAMBOISE instance unbundling process is depicted in Figure 5.5; it starts

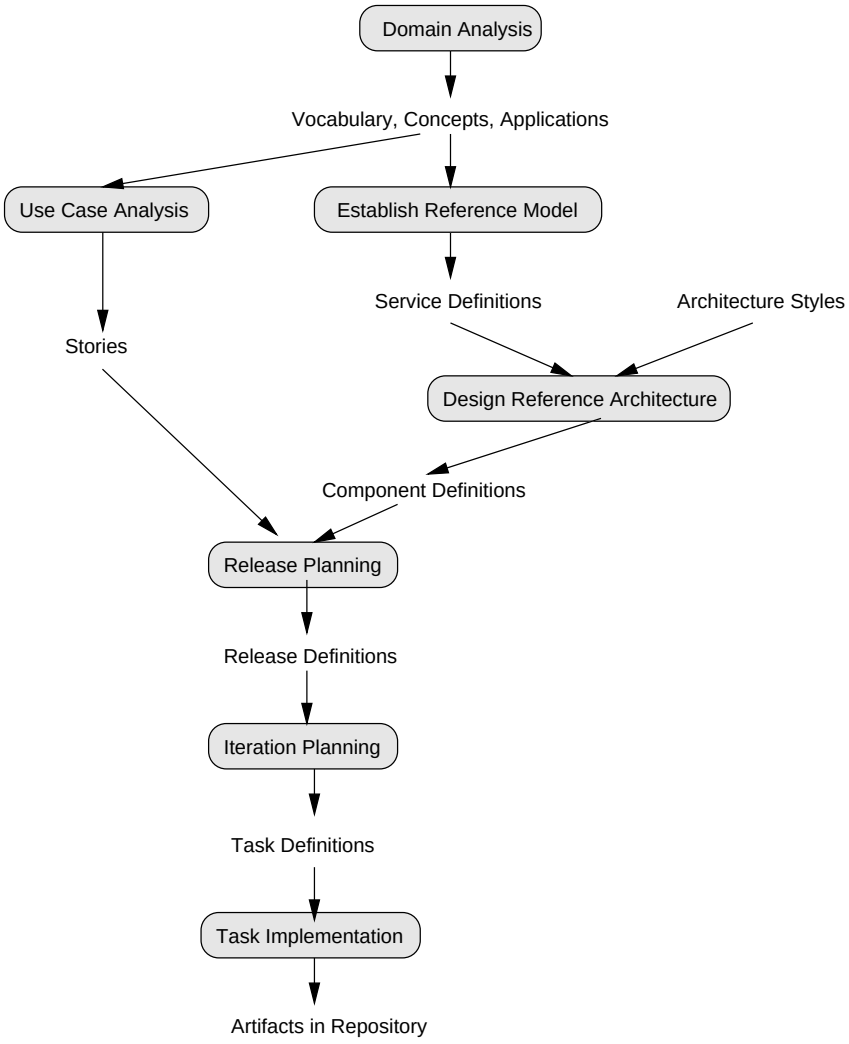

Figure 5.5: The FRAMBOISE Instance Unbundling Process

with a domain analysis that yields the vocabulary and concepts of active database technology as well as the initial applications. The domain analysis enables on the one hand

the developers to identify and describe the stories that are implemented with the unbundled components. On the other hand the domain analysis is a basis to design the reference architecture of the ECASs. Component frameworks incorporate architectures of the systems to be built by means of them [Szy97]. In fact these architectures are reference architectures as discussed in Section 4.2.3. Even though architecture design appears to be an issue of development *with* reuse (cf. Fig. 5.2 and 5.3) a reference architecture must be designed during the unbundling process. Decomposing a system into reusable parts obviously begins at an architectural level and it is unthinkable to develop reusable components without having any architectural vision in mind.

Henceforth, after the stories are described and the reference architecture is designed, release planning is performed. Releases are basically defined according to XP practice where the customer "picks the most valuable stories and groups them into releases" [Bec00]. Release planning is, however, adjusted according to the given facts of the reference architecture. That means an exclusively business-oriented priority list may be overruled to be better able to generalize specific aspects. Upon release planning the further unbundling process proceeds according to the common practices of extreme programming, i.e., releases are broken down into iterations which are themselves split into program tasks etc.

Note that the adoption of extreme programming implies that the unbundling process incorporates also feedback loops which are not depicted in Figure 5.5. Continuous refinement of the design and insights gained when a task is implemented eventually lead to refactoring of code blocks, components or even of the reference architecture. XP does not prescribe a well-defined procedure to perform such feedback loops. Instead, changes are principally coped with when they are detected, whereas the practices of extreme programming (continuous integration and testing, collective ownership of code etc.) ensure a stable development progress. We refrained ourselves therefore from overcrowding Figure 5.5 with myriads of backward pointing arrows that clarify nothing but confuse the reader.

5.3 The FRAMBOISE Bundling Process

The instance rebundling process in FRAMBOISE represents in its essence application engineering as it is outlined in Section 5.1.2. Hence the FRAMBOISE bundling process is also centered around the spiral model according to the suggestion of [Sam97] (cf. Sec. 5.1.2). In the context of rebundling, we identify the following modifications which are attributed to the various quadrants of Figure 5.4:

- *Quadrant I:* The first quadrant of the spiral involves determining the objectives of the system to be developed, identifying realization alternatives and constraints imposed on them. Determining the objectives to be developed means in our context that the ADBI must identify the rule and knowledge model of the future ECAS. On the one hand these models may be derived from a specific ADBMS

proposal if such a system should be provided. On the other hand, it is conceivable that active database functionality should be provided for specific application systems. In that case the ADBI must beforehand identify which aspects of the overall system should be realized by means of ECA rules in order to determine the knowledge and rule model of the ECAS. Hence some aspects of rule development (i.e., the so-called *rule requirement extraction* or for short *rule extraction* [Vad99]), must precede the actual rebundling steps.

The constraints imposed on the rebundling activity are on the one hand the opportunities offered by the underlying DBMS to interact with the ECAS. On the other hand there are timely and financial restrictions which may subsequently enforce the selection of an alternative (e.g., it may be prohibitive to perform an unbundling task so that an ADBI chooses an alternative relying on a downsized knowledge model that renounces on several active features). Assuming that the construction of an ECAS is not discarded at all, the result of quadrant I is the specification of at least one ECAS with the identification of the candidate components.

- *Quadrant II:* In the second quadrant, the ADBI evaluates the options and their risks. The result of these activities is the decision on which ECAS is realized and which components are reused, respectively what modifications/adaptations are to be made.

- *Quadrant III:* The third quadrant of the spiral comprises the development of the next-level product. That means that the selected components have to be modified/adapted and integrated into a skeleton of the future ECAS.

  The result of this quadrant is the provision of an operational ECAS.

- *Quadrant IV:* The fourth quadrant of the spiral involves planning for the next phases. The tasks of this quadrant involve "debriefing" activities such as the incorporation of novel components into the repository as well as the evaluation of components for their reuse potential and reporting of reuse experiences.

The instance bundling process is illustrated in Figure 5.6. For the sake of clarity, the instance bundling process is depicted as a sequence of activities. However, the actual process control is performed according to the spiral model as discussed above. the component engineering activity in Figure 5.6 involves black box (which includes minor adaptations like parameterization) as well as white box reuse. According to the complexity of the problem to be solved, white box reuse may result in an further unbundling step which is covered by the above instance unbundling process.

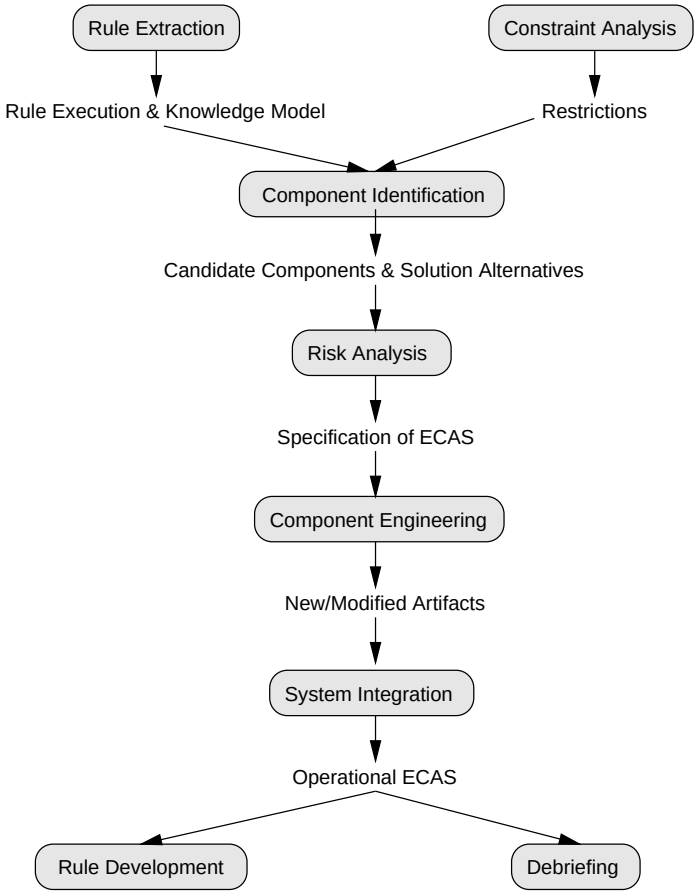

Figure 5.6: The FRAMBOISE Instance Bundling Process

5.4 Conclusion

In this chapter the procedures to furnish FRAMBOISE with an initial set of components and to build subsequently specific ECA Systems have been identified. The courses of action described above are conceived as a basis to establish full-fledged process definitions in a productive environment.

Thus, all preliminaries to effectively unbundle ADBMSs have been achieved according to the meta process established in Section 3.5.3. It is therefore feasible to define the reference architecture of the ECA Systems in the next chapter.

Chapter 6

The Reference Architecture of ECA Systems

This chapter describes the systematical architecture design of the ECA systems in detail and comprises therefore one of the chief contributions of this thesis. Regarding the state of the art in the domain of ADBMSs, a situation where one can consult an authoritative source such as a textbook, adopt the solution to establish the reference architecture and press on is far away. Even though researchers have developed a wide range of prototype active database systems including a variety of architectural concepts, a distinct perception of the architecture of active database systems did not result so far, let alone of a concept suited for a component-oriented approach like FRAMBOISE. Since any ADBMS must provide facilities to manage rules as well as to detect events and execute rules, many of the suggested architectures coincide in some of their chief elements. However, these proposals still vary considerably, because they are typically streamlined to enable the implementation of the respective prototype. As a consequence it is necessary to design the reference architecture of the ECASs according the procedure described in Section 4.2.3 (Fig. 4.3).

The chapter is organized as follows: Section 6.1 develops a reference model of active database functionality suited to unbundle ADBMSs. This reference model is subsequently mapped in Section 6.2 to an architecture style that settles the central theme of the reference architecture. Section 6.3 concludes the chapter.

6.1 A Reference Model of Active Database Functionality

According to the definition of a reference model (cf. Section 4.2.3), this section identifies a functional decomposition of ADBMSs into parts that cooperatively provide active database functionality. The starting point is the assumption discussed in Section 2.4 that any ADBMS must provide means to define and manage rule definitions, to detect events and to execute rules whereas these facilities must interoperate with the

"traditional" DBMS services as shown in Figure 2.3.

The reference model is elaborated from two points of view: First in Section 6.1.1 the functional decomposition of the major subsystems of Figure 2.3 is discussed. This implicitly assumes a rather tight coupling between the active and the passive DBMS parts. Afterwards, Section 6.1.2 examines which parts are additionally required when active database functionality is conceived as a service that is decoupled from a DBMS.

6.1.1 The Main Parts of an ADBMS

The main parts of an ADBMS are grouped in two categories, i.e., one for rule management and another for rule execution which are in turn discussed subsequently.

Rule Specification and Registration

In this paragraph the basic functions involved in the specification of rules as far as the ADBMS proper is concerned (i.e., design tools providing rule specification facilities are not considered) and their registration within the system are identified. The registration of a new rule in a rulebase requires a series of administrative processes for which the respective facilities must be provided. They are schematically depicted in Figure 6.1. Rules are usually specified by means of a rule language that is processed through a specific compiler. The front end of such a compiler typically performs lexical and syntax analysis whereas its backend performs the semantic analysis as well as the code generation. Code generation must be seen in this case as the extraction of the event, condition and action definition as well as of control information like coupling modes, priorities etc. and their mapping to rule creation commands for the rule management subsystem. Code generation causes in turn that the respective definitions are stored persistently in the rulebase, initializes the event detection mechanism if the respective event is not already known to the system, and compiles and links condition and actions and stores them within the condition/action library which is considered as a special compartment of the rulebase.

The Rule Execution Subsystem

Rule execution involves all activities from event detection until the eventual invocation of DBMS commands or external programs as a result of action execution as shown in Figure 6.2. Event detection subsystems are typically split into event detectors and event handlers. Event detectors are those elements that detect the occurrence of an event in the sense of the semantics of the event definition. The identification of the rule(s) that are triggered can either be done by a specialized event handler that knows locally what rules are to be fired once the event has been raised or through the use of a separate registration mechanism and an additional lookup. Furthermore event handlers typically log all event occurrences and their parameters. Event logging as well as event composing can be a major performance factor in ADBMSs.

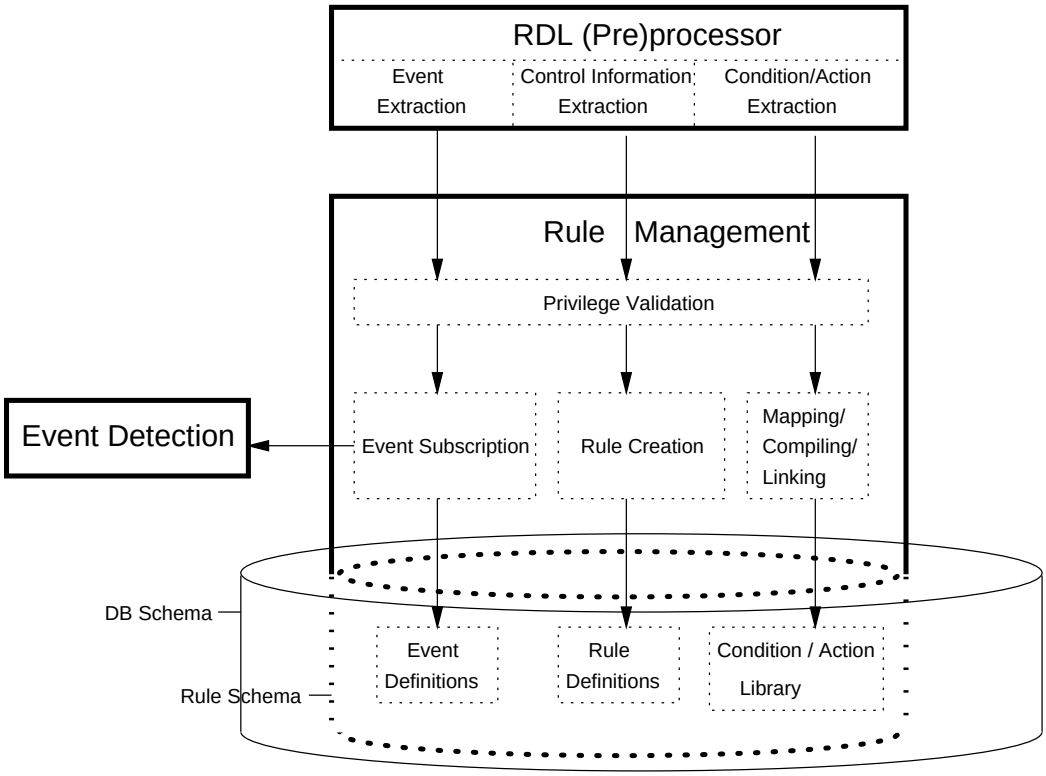

Figure 6.1: Rule Registration Subsystem of an ADBMS

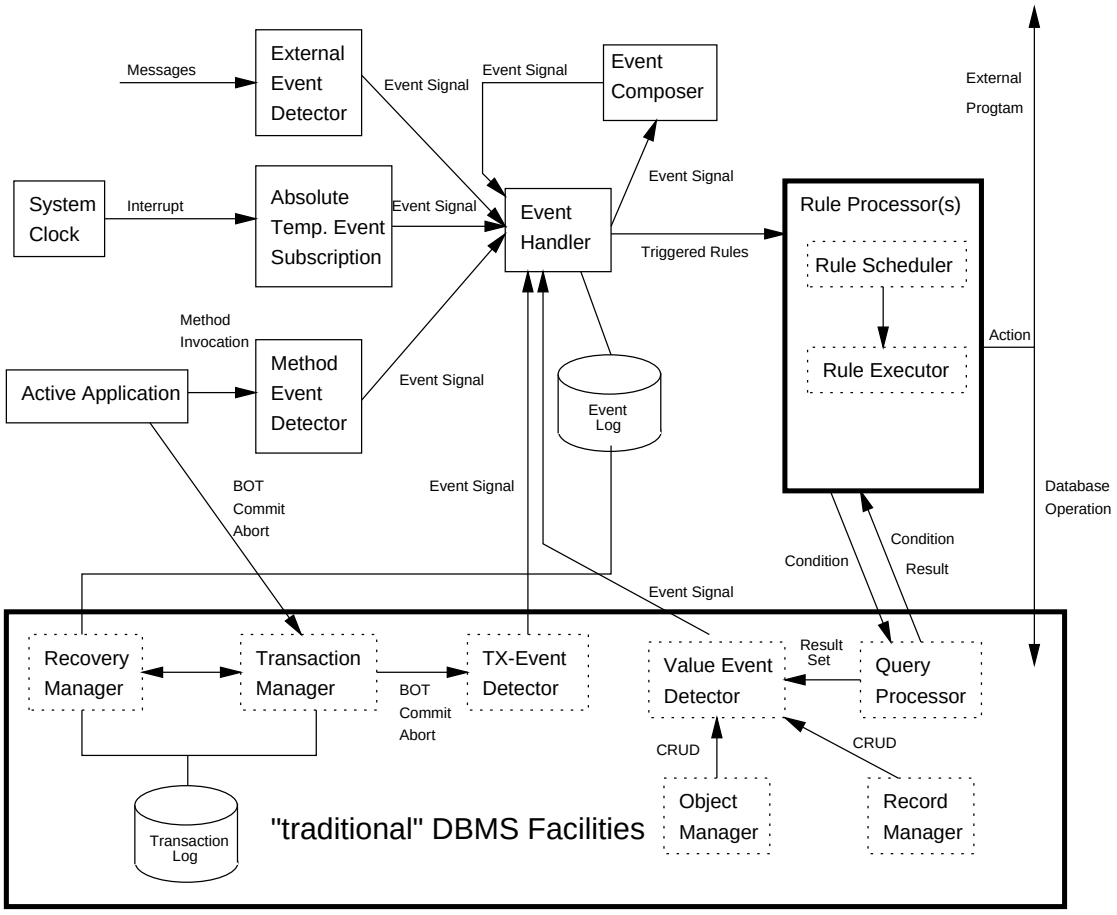

Figure 6.2: Rule Execution Subsystem of an ADBMS

The rule processor can be roughly subdivided in a rule scheduling mechanism determining the next rule instances to be executed and a specific executor that invokes the respective conditions and actions.

Rule scheduling and execution is highly dependent on the DBMS's transaction management and must be synchronized with the execution of user transactions. How user transactions and rule execution are synchronized depends on the transaction model of the DBMS, the interfaces to the transaction manager provided by the DBMS and the way the ADBMS takes advantage of the provided facilities.

Figures 6.1 and 6.2 are related twofold: On the one hand, the event detectors that belong actually to the rule execution subsystem are informed by the rule registration subsystem about the events they have to monitor. On the other hand, the event handler identifies the rules to be triggered by querying the rulebase which is part of the rule management subsystem. For the sake of readability, these interdepedences are omitted in Figure 6.2.

The "active" (w.r.t. database functionality) and the "passive" parts of a rule execution subsystem are typically closely coupled. This is illustrated by drawing some event detectors in Figure 6.2 within the "traditional" elements. That means that they are so closely related that it is quite impossible to distinguish where the passive part and the active begins.

On the one hand this tight coupling is necessary to be able to perform active database behavior at all (e.g., detection of database events must take place within the passive DBMS). On the other hand such a close coupling is sometimes also necessary to perform with the required runtime efficiency.

6.1.2 Separating the Active Mechanisms from the DBMS

Some the active database projects anticipated the necessity to build ADBMSs at least partially out of preexisting parts e.g., as layer on top of a passive DBMS [GGD+95b] or by implementing the ADBMS as an extension of a database construction toolkit [CKTB95, BZBW95]. Nevertheless, none of these architectures has been designed for a reuse-oriented or even component-based approach so that ADBMSs still tend to be monolithic software systems.

The first ideas to separate active database functionality from the DBMS in order to provide so-called *activity components* were discussed in [GKvBF98]. A series of consecutive unbundling steps sketch various generic architectural proposals that can be used as a starter-kit to initiate the unbundling of active database mechanisms for a specific purpose [KGvBF98]. Figure 6.3 depicts an unbundling step separating the active functionality from the DBMS as an *activity service* that includes rule management and rule execution. Since rule management can exploit database services to store the rule base persistently and rule execution requires the ability to detect database events, to evaluate conditions, and to execute actions both chief tasks of an activity service are related to passive database functionality. The passive database manager is involved in

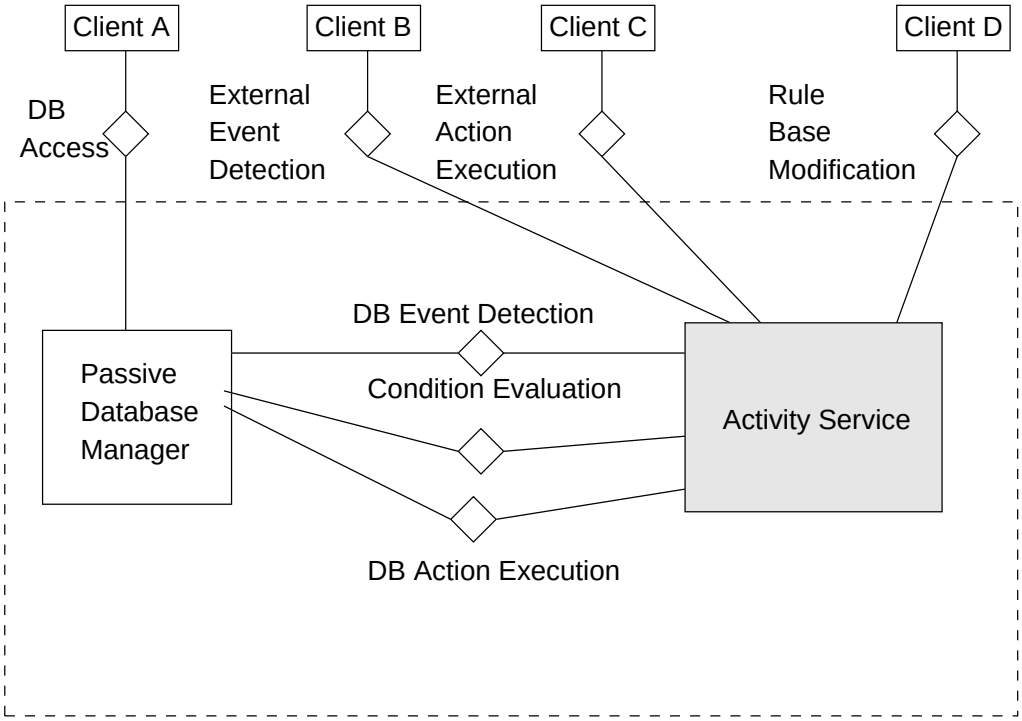

Figure 6.3: The Reference Model of Unbundled Active Database Systems.

rule execution as an important event source, as well as to evaluate conditions that refer to database states and execute actions modifying the database.

Figure 6.3 focuses on the connection mechanisms that must be provided in order to furnish a stand-alone active database service. These mechanisms – depicted as diamonds[1] in Figure 6.3 – are considered as "fat connectors" that provide substantial functionality in order to mediate the respective interaction.

Hence 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.

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 activity service. It determines what type of events the DBMS produces and to what events the event service subscribes to. Ac-

[1]Architectures consisting of components and connectors can be described by means of ER-style diagrams, denoting components as rectangular boxes and connectors as diamond-shaped relationships.

cording 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 activity service cooperates with the database manager via the *condition evaluation* and the *database action* execution connectors. These connectors require database access capabilities as provided by the database access connector[2]. Moreover, they require a transaction service allowing database accesses to be performed as part or subtransaction of a transaction performed on behalf of some client (at least if all coupling modes are to be supported). If the database manager requires that access commands are statically compiled before they are executed (as opposed to dynamic invocation), the corresponding compilation is also performed via the condition evaluation or database action execution connector during a rule base modification. Furthermore, the activity service may also make direct use of the database access connector in order to store and manage rules persistently (not shown in the Figure 6.3).

6.2 The Design of the Reference Architecture

The most obvious approach to design the reference architecture of the ECASs were to take the components of our reference model of ADBMSs "as is" and couple them by means of so-called *event-based implicit invocation*. The idea behind the implicit invocation architecture style is that instead of invoking services provided by other components directly (e.g., as a procedure call), a component announces one or more events. Other components in the system can register in an event by associating a procedure with an event. When the event is raised, the system invokes all of the services that have been registered for the respective event. Thus an event announcement implicitly causes the invocation of procedures in other modules. This architecture style promotes loose coupling because any component can be introduced into a system simply by registering it for the respective events. Hence the implicit invocation style strongly supports reuse and eases system evolution [SN92].

However, the drawbacks of this architecture style make it unsuitable for FRAMBOISE. The primary disadvantage of implicit invocation is that components basically relinquish control over the computation performed by the overall system. A component that announces an event cannot rely on the order the components interested in this service are invoked nor does it know when all invocations associated with a specific event are finished. Also, reasoning about correctness can be problematic [GS94], in contrast to traditional invocation mechanisms like procedure calls, where one needs

[2]The database access connector allows clients to establish a session with the database manager, to start and end transactions, and to access or modify data items (e.g. relations, objects) within the database. Thereby, the database manager synchronizes concurrent client accesses.

only consider a procedure's pre- and postcondition when reasoning about an invocation of it.

An architecture style that pervades the reference architecture of ECASs must promote concise reasoning about correctness in a better way than implicit invocation. It is of utmost importance that the construction process of FRAMBOISE asserts that the rule execution behaves in accordance with the semantics defined by the rule and rule execution model of the respective ECAS. Thus the adoption of architecture styles that structure the ECASs more concisely and prescribe clear patterns of interaction among the components are preferred.

We design the reference architecture of the ECASs in a top-down manner, i.e., by considering the ECAS as a black box that is decomposed subsequently into components and subcomponents. In a preliminary step the basic interaction of an ECAS with its environment is analyzed in Section 6.2.1. Subsequently Section 6.2.2 introduces the architecture style that forms the underlying theme of the reference architecture. The constituents of this elements of this preliminary architecture are decomposed in Section 6.2.3 to achieve the actual reference architecture.

6.2.1 ECA Systems as Monolithic Service

The reference model of a standalone activity service depicted in Figure 6.3 corresponds to a large extent to our conception of ECA Systems. An ECAS embraces the functionality of an activity service as well as that of the various diamond shaped connection mechanisms (e.g., database action execution connector). However, the latter are considered as components of an ECAS, because it is misleading to address them in our context as connectors. These facilities are henceforth called *adapters*.

Even though Figure 6.3 gives a good intuitive picture of the problems to be coped with, it leaves a number of fundamental questions unanswered. For instance this setup does not indicate whether an activity service is a rule server for the passive DBMS or whether it uses the DBMS as server to execute (database) actions. It is therefore wise to specify an ECAS beforehand as a monolithic entity in order to analyze how an ECAS interacts with its environment. The architecture definition language WRIGHT [AG94, AG97, All97] (cf. Sec. 4.2.4) supports this procedure by permitting an *incremental* approach to formalize a system. It is feasible to specify some of the architecture and get immediate, valid results, without incurring the cost of a full specification. Then one can add more to the specification and get more results without having to redo what has been already accomplished.

First, two WRIGHT interface types[3] are established. They define the "standard" behaviour of a client that pulls requests from a server and its counterpart, i.e., a server that pushes the results of a request to a client:

Interface Type IServerPush = $open \rightarrow Operate \;\square\; \S$
  where Operate = $request?x \rightarrow \overline{result!y} \rightarrow Operate \;\square\; close \rightarrow \S$

[3] A description of the Architecture Definition Language WRIGHT is given in Appendix A

Interface Type IClientPull = $\overline{open} \rightarrow Operate \sqcap \S$
where Operate = $\overline{request!x} \rightarrow result?y \rightarrow Operate \sqcap \overline{close} \rightarrow \S$

The following connector regulates a typical client-server interaction between two components obeying the above interfaces:

Connector ClientServer =
Role Client = IClientPull
Role Server = IServerPush
Glue = $Client.open \rightarrow \overline{Server.open} \rightarrow Glue$
$\Box\ Client.close \rightarrow \overline{Server.close} \rightarrow Glue$
$\Box\ Client.request \rightarrow \overline{Server.request} \rightarrow Glue$
$\Box\ Server.result?x \rightarrow \overline{Client.result!x} \rightarrow Glue$
$\Box\ \S$

ECAS are specified as shown in Figure 6.4 as a single WRIGHT component with several ports through which the ECAS sends or receives information and requests. The

Component ECAS(nEvSrc: 1 . . . ; nEvCh: 1 . . . ;
nOpDef: 1. . . ; nOpChnl: 1. . . ; nRbClients: 1. . . ; C: Computation)
Port $\text{RbAccess}_{1...nRbClients}$ = IServerPush
Port $\text{EventDef}_{1...nEvSub}$ = IClientPull
Port $\text{EventChnl}_{1...nEvSrc}$ = IServerPush
Port $\text{OpDefs}_{1...nOpDef}$ = IClientPull
Port $\text{OpChnl}_{1...nOpChnl}$ = IClientPull
Computation = C

Figure 6.4: The WRIGHT Specification of an ECAS

following ports are defined:

- The `RbAccess` port enables an external client to access the rulebase of an ECAS, whereby the ECAS acts as a simple server. Since various clients can access the rulebase at the same time, this port is parameterizable with an integer number, expressing the actual number of identical *RbAccess* ports.
- Through the `EventChnl` ports – addressed as *event channels* – the ECAS receives event signals from event sources (external ones as well as the DBMS). The ECAS acts again as a server, i.e., it receives event signals and has to process eventually associated rules. As a result of the invoked server returns the

signalling process to proceed, unblocks the triggering transaction etc. The event signals and the results sent through a channel are transmitted in the proprietary format of the respective event source whereby they are translated to and from an ECAS internal format by the specific event adapters which are situated *within* the ECAS. Thus it is not necessary to distinguish at the architectural level between different type of event channels (i.e., such for the DBMS and such for external event sources) but we can focus on the interaction pattern.

- By means of the `EventDef` ports an ECAS accesses event sources to subscribe for the detection of specific event occurrences. For instance the definition of an event to monitor updates in a database might imply that a corresponding trigger is defined in the DBMS. Hence a (DBMS-specific) trigger definition command is sent over the associated event definition channel. Since the ECAS requests an event source to monitor an event occurrence, the communication along these channels obeys a client-server pattern whereby the ECAS represents the client.

- Finally there are the channels to define conditions and actions and to invoke these operations subsequently. These ports are addressed as `OpDef` and `OpChnl`, whereby the former are used to transmit commands to compile and link operations and the latter – the *operation channels* – enable the invocation of conditions and actions respectively. In contrast to the reference model depicted in Figure 6.3 there is no distinction between operation ports that interact with a DBMS and those communicating with an arbitrary external event source, because they obey all the same interaction patterns. Again there are ECAS internal adapters providing the commands in a format understandable for the respective device.

- The computation part is simply described with the parameter `C` which acts as a *placeholder* which is filled with a parameter when the ECAS is instantiated in an actual configuration. Thus it is feasible to express different ECA Systems with the same component description, whereby the actual instantiation of `C` represents the behaviour of the respective ECAS. The parameter `C` embraces also the subsequent decomposition of the ECAS into subcomponents computation part as it is feasible to instantiate `C` with a configuration representing the architectural subsystems. Hence this nested description has an associated set of bindings, which define how the unattached port names on the inside subcomponents are associated with the associated port names of the ECAS components.

Note that it were in principle feasible to summarize the rulebase access, event definition and operation ports as one single port type without formally losing information since these ports obey all the `IClientPull` interface. However the computation parts became too complex in that case. Thus we prefer to divide into the above broad categories. Nevertheless, an adapter is not necessarily related to exactly one port but it is conceivable that several ports are associated to one adapter. For instance, a DBMS adapter might provide one port to define triggers (programming the event detector),

another port to define and compile stored procedures, one channel to send event signals and several channels to receive action commands in order to enable parallel rule execution. Furthermore, ports may be multiplexed by specific connectors, so that their behaviour appears to the outside as one single channel.

For an ECAS component conceived as monolith, the computation part can be defined as a single CSP process that performs two main tasks. On the one hand it listens at every event channel for event signals to perform the rule execution and on the other hand it processes rulebase modification commands that are invoked via the `RbAccess` port. These tasks are multithreaded activities (expressed by the CSP Operator $\|$ that describes concurrent processes) as an ECAS must be able to process several events and rulebase modification commands simultaneously.

Computation MonolythicECAS(nEvSrc: 1 ...; nEvCh: 1 ...; nOpDef: 1...;
nOpChnl: 1...; nRbClients: 1...;) =
$\Big\|\ \forall\, i \in nEvSrc \Big\|\ EventChnl_i.open \rightarrow Listen_i$
$\Big\|\ \forall\, j \in nRbClients \Big\|\ RbAccess_i.open \rightarrow ModifyRulebase_j \ \Box\ \S$
where Listen_i = ...
where ProcessRules_i = ...

Once an event channel *i* is opened the ECAS listens at the respective port for incoming event signals or the command to close the channel as expressed by the process $Listen_i$.

$$\text{Listen}_i = EventChnl_i.signal?request \rightarrow (ExecRules_i \sqcap Listen_i) \ \Box\ EventChnl_i.close$$

Both CSP events[4], i.e., signalling an event or closing the channel are equally possible and are initiated by components residing outside of an ECAS (i.e., the CSP events are written without an overbar). According to the consistency rule 5 (initiator commits cf. Appendix A.3) an ECAS must be able to agree with both CSP events which is ensured by the application of the external choice operator $\Box$. Once an event is signalled rule processing takes place (defined in the CSP process $ExecRules_i$) if there are any rules associated for this event or the ECAS listens again to further event signals.

Note that the treatment of incorrect commands (e.g., an incorrect rule definition) is not modeled at this level of abstraction. Concerning the specification of a reference architecture it is sufficient to think for instance in terms of a rulebase administrator trying out successive communications until finding the correct ones. It is also safe to assume that signalled event occurrences are always matched by corresponding event definitions in the rulebase. In that sense, any activities where the rulebase is queried during the rule execution process are skipped for the time being. Instead they are modeled as an internal (w.r.t. the ECAS) respectively non-deterministic choice $\sqcap$ whether any rule

[4]CSP uses the notion "event" as a fundamental concept of the formalism. In order to avoid a confusion with the event notion applied in ADBMSs we refer to events in CSP exclusively as *CSP events* whereas active database events are simply referred to as events.

processing is initiated through an event occurrence or not. Using non-determinism to elide details provides a nice balance between complexity and fidelity. It reduces complexity, because no state must be maintained to indicate which subset is chosen but it indicates that a choice is made. The correct choice will be a refinement of this abstract specification and could be specified and analyzed in a more detailed specification.

The CSP process representing the rule execution, i.e., $ExecRules_i$, consists basically of two consecutive operations which stand for the condition evaluation and action execution. Both processes can be represented by the same CSP process $OpExe_{j,x}$ whereby the index j is instantiated with the channel number and x with the result value of the operation.

$$\text{ExecRules}_i = \exists\, j \in 1 \ldots nOpDef \bullet OpExe_{j,x};\ \exists\, k \in 1 \ldots nOpDef \bullet OpExe_{k,x};$$
$$Listen_i \not< x \neq \emptyset \not> \overline{EventChnl_i.result!x} \rightarrow Listen_i$$

The semicolon operator `;` allows to form a sequence of processes whereas the operation $A \not< cond \not> B$ describes a choice between two alternatives that is based on a condition *cond*. Option *A* is chosen if the condition holds otherwise the process behaves according to alternative *B*. Thus *ExecRules* begins with the execution of an operation on a channel j, X associated to the respective "active database" condition and if this evaluates to true (expressed through the statement $\not< x \neq \emptyset \not>$ it executes the corresponding action (represented by another process *OpExe* and listens subsequently to further event occurrences. Otherwise the CSP process continues with a return CSP event in order to comply with the interface definition `IServerPush` to ensure port computation consistency (cf. consistency rule 1) and behaves subsequently again according to the CSP process *Listen*. The CSP process representing the execution of an action or the evaluation of a condition is defined as follows:

$$\text{OpExe}_{n,x} = \overline{OpChnl_n.open} \rightarrow \overline{OpChnl_n.request!y} \rightarrow OpChnl_n.result?x \rightarrow$$
$$\overline{OpChnl_n.close} \rightarrow \overline{EventChnl_i.result!x}$$

The other core activity of an ECAS, i.e., the modification of a rulebase, implies either the subscription of a new event definition over a specific event definition port or the registration of a condition (or an action respectively) along an `OpDef` port.

$$\text{ModifyRulebase}_j = RbAccess_j.close \,\square\, (RbAccess_j.request \rightarrow$$
$$\rightarrow (SubscribeEvent \sqcap RegisterOperation \sqcap ModifyRulebase))$$
where SubscribeEvent $= \exists\, i \in 1 \ldots nEvSrc \bullet \overline{EventSrc_i.open} \rightarrow \overline{EventSrc_i.request}$
$\rightarrow EventSrc_i.result?x \rightarrow \overline{EventSrc_i.close} \rightarrow \overline{RbAccess_j.result!x} \rightarrow ModifyRulebase_j$
where RegisterOperation $= \exists\, i \in 1 \ldots nOpDef \bullet \overline{OpDefs_i.open} \rightarrow \overline{OpDefs_i.request}$
$\rightarrow OpDefs_i.result?x \rightarrow \overline{OpDefs_i.close} \rightarrow \overline{RbAccess_j.result!x} \rightarrow ModifyRulebase$

In order to analyze the interaction between an ECAS and the DBMS it is necessary to describe the assumed behavior of a DBMS in WRIGHT. Nowadays commercially available DBMSs typically apply the so-called *publish and subscribe technique* to notify external clients about database events. That means such DBMSs allow an external system to subscribe to event occurrences whereby these clients can also specify a range of events to be supervised (e.g., for entities in a specific segment, table or even specific objects). Henceforth these clients are notified upon the occurrence of these events until they unsubscribe. Event notification is transmitted *asynchronously*, i.e., the respective database operations continue upon event signalisation without giving the observers a chance to block the triggering transaction. This behaviour is described with the following WRIGHT interface definition:

Interface Type IAsynchNotifier = $open \rightarrow Notify \;\square\; \S$
  where Notify = $\overline{raise!event} \rightarrow Notify \sqcap close \rightarrow \S$

The CSP process described by the `IAsynchNotifier` interface differs from the event channels of an ECAS obeying the `IServerPush` protocol. Both interfaces do not initiate the process but expect to be opened from external entities. Hence these two processes would not agree who initiates the interaction which in turn leads to a CSP deadlock[5]. Furthermore they have a different alphabet (besides the events `open` and `close`). Thus we have to define a connector between an `IAsynchNotifier` and an `IServerPush` that ensures the absence of a local deadlock (consistency rules 2 and three) and ensures that there is exactly one initiator of the communication (consistency rule 4).

This is feasible regarding that an ECAS accesses event sources to subscribe for the detection of specific event occurrences by means of the `EventDef` ports. Thus we can define a connector that multiplexes an `EventChnl` and `EventDef` port to interact with an `IAsynchNotifier`.

Connector AsynchNotifier =
  Role Notifier = IAsynchNotifier
  Role EventDef = IClientPull
  Role EventChnl = IServerPush
  Glue = $EventDef.open \rightarrow \overline{EventChnl.open}$
    $\rightarrow \overline{Notifier.open} \rightarrow Operate \;\square\; \S$
    where Operate = $Notifier.raise?event \rightarrow \overline{EventChnl.request!event} \rightarrow Operate$
      $\square\; EventChnl.reply?event \rightarrow Operate$
      $\square\; EventDef.close \rightarrow \overline{Notifier.close} \rightarrow \overline{EventChnl.close} \rightarrow \S$

At the architecture level the interest is only on the basic communication between the DBMS and the ECAS when event signalling and rule execution takes place. It

[5]Note that in WRIGHT, deadlock is used simply to indicate some kind of problem in the specification and cannot be taken to mean that the system will actually halt.

is therefore sufficient to assume a DBMS as a WRIGHT component with a number of identical ports that provide database sessions for external clients as well as ports to signal notifications about database events. The following component stands for a typical passive DBMS providing asynchronous event notification:

Component VanillaDBMS(nClients: 1 . . . ; nListeners: 1 . . .)
Port $\text{DBClient}_{1\ldots nClients}$ = IServerPush
Port $\text{Listener}_{1\ldots nListeners}$ = IAsynchNotifier
Computation = $\forall\, i \in 1 \ldots nClients \,\|\, DBClient_i.open \rightarrow Operate_i \sqcap \S$
$\forall\, j \in 1 \ldots nListeners \,\|\, Listener_j.open \rightarrow Notify_j \sqcap \S$
where Operate_i = $DBClient_i.request \rightarrow \overline{DBClient_i.result!x}$
$\rightarrow Operate \,\square\, \overline{DBClient_i.close} \rightarrow \S$
where Notify_j = $\overline{Listener_j.signal!event} \rightarrow Notify_j \,\square\, Listener_j.close \rightarrow \S$

In order to describe a system architecture, the components and connectors of a WRIGHT description are combined into a *configuration*. Figure 6.5 describes the architecture of a system where a monolithic ECAS interoperates with a passive DBMS: A configuration starts with the definition of the interfaces, components and connectors (for the sake of clarity the complete description is not repeated here) which can be considered as type definitions. In the subsequent compartment the *instances* of these definitions are declared. Thereby the parameters are bound to specific values. Finally, the configuration is completed by describing the *attachments*. They define the topology of the configuration by specifying which components participate in which interactions. This is done by associating a component's port with a connector's role. In the above example a simple configuration is assumed where the ECAS interacts exclusively with the DBMS and one rule client. The DBMS interoperates with only one additional DBMS client. There are components defined that stand for the DBMS and the rulebase client, because these components are outside of the system under consideration. The client role of the connector `dbSession` and `rbCS` sufficiently describe the behaviour of these components. This configuration obviously is a somewhat simplistic view on an effective situation. However, we experienced it as an appropriate abstraction for a human designer to grasp the interactions among an ECAS and its environment at an architectural level.

In contrast to the above specified passive DBMS, a hypothetical DBMS that cooperates with an external rule processing facility hands the control to this service until all rules are processed. This behaviour is sketched by means of the following "semi passive" DBMS component:

Component SemiPassiveDBMS(nClients: 1 . . .)
Port $\text{Client}_{1\ldots nClients}$ = IServerPush
Port RuleSystem = IClientPull
Computation = $\overline{RuleSystem.open} \rightarrow Operate$

Configuration SimpleConfig
  Component VanillaDBMS(nClients: 1 . . . ; nListeners: 1 . . .)
  Component ECAS(nEvSrc: 1 . . . ; nEvCh: 1 . . . ;
    nOpDef: 1. . . ; nOpChnl: 1. . . ; nRbClients: 1. . . ; C: Computation)
  Computation MonolythicECAS(nEvSrc: 1 . . . ; nEvCh: 1 . . . ;
    nOpDef: 1. . . ; nOpChnl: 1. . . ; nRbClients: 1. . . ;)
  Connector ClientServer
  Connector AsynchNotifier
Instances
  ecas: ECAS(1;1;1;1;1;MonolythicECAS(1;1;1;1;1))
  dbms: VanillaDBMS(3,1)
  dbSession,cs1,cs2,rbCS: ClientServer
  dbEvents: AsynchNotifier
Attachements
  dbms.DBClient$_1$ **as** dbSession.Server
  ecas.OpDefs$_1$ **as** cs1.Client
  dbms.DBClient$_2$ **as** cs1.Server
  ecas.OpChnl$_1$ **as** cs2.Client
  dbms.DBClient$_3$ **as** cs2.Server
  dbms.Listener$_1$ **as** dbEvents.Notifier
  ecas.EventDef$_1$ **as** dbEvents.EventDef
  ecas.EventChnl$_1$ **as** dbEvents.EventChnl
  ecas.RbAccess$_1$ **as** rbCS.Server
End SimpleConfig

Figure 6.5: Configuration of a monolithic ECAS and a passive DBMS

where Operate = $\forall\, i \in 1 \ldots nClients \,\|\, DBClient_i.open \rightarrow DBSession_i$
where DBSession$_i$ = $DBClient_i.request \rightarrow$
$(\overline{DBClient_i.result!x} \sqcap \overline{RuleSystem.request!event_i} \rightarrow$
$RuleSystem.result?event_i \rightarrow \overline{DBClient_i.result!x})$
$\Box\ DBClient_i.request.close$

Note that this "semi passive" DBMS initiates the communication between the DBMS and the rule system because the DBMS relies on the latter's availability to ensure correct behaviour. Thus the connector regulating the interaction between the DBMS and the ECAS must not multiplex the `EventDef` and the `EventChnl` in order to avoid a deadlock. In fact, a simple client-server connection between the `RuleSystem` port of the DBMS and an `EventChnl` port of an ECAS is absolutely sufficient.

The WRIGHT definitions established so far describe the high-level design of an ECAS by defining the fundamental computation patterns as well as the basic interaction protocols between an ECAS and its environment in a concise way. Thus the abstractions and intuitions of the informal model established in Section 6.1.2 have been made explicit and the subsystem design can begin.

6.2.2 The Gross Structure of an ECAS

This section describes the first decomposition step. First we describe how an architecture style is applied to settle the basic structure of the reference architecture. Subsequently we illustrate how the decomposition of an ECAS is expressed in WRIGHT.

ECA Systems as Virtual Machines

The design of the reference architecture of ECASs is based on the so-called *virtual machine architecture style* [BCK98, GS94], an architecture pattern which is defined as follows:

Problem: Virtual machines are software systems that simulate some functionality that is not native to hardware and/or software on which it is implemented. Hence this style is adequate for applications in which the most appropriate language or machine for executing the solution is not directly available.

Context: Virtual machines are most often designed to bridge the gap between a desired machine or language and some machine or language already supported by the environment.

Solution:

- *System Model:* Interpretation of a given program according to specific input. The virtual machine style is synonymously addressed as *table driven*

interpreter architecture style (or pattern respectively). We do not apply the latter term in order to avoid a confusion with the differently structured interpreter design pattern introduced in [GHJV95].

- *Components:*
  - The execution engine that interprets and executes the program statements.
  - The representation of the virtual machine's control state (e.g., values of registers or the current statement to be executed).
  - A compartment that contains the program to be carried out by the execution engine.
  - A representation of the current state of the program to be interpreted (e.g., the values of variables assigned during program execution).
- *Connectors:* Data access and procedure calls.
- *Control Structure:* Usually state-transition graphs for execution engine; input-driven for selection of what to interpret.

Diagram: According to Figure 6.6

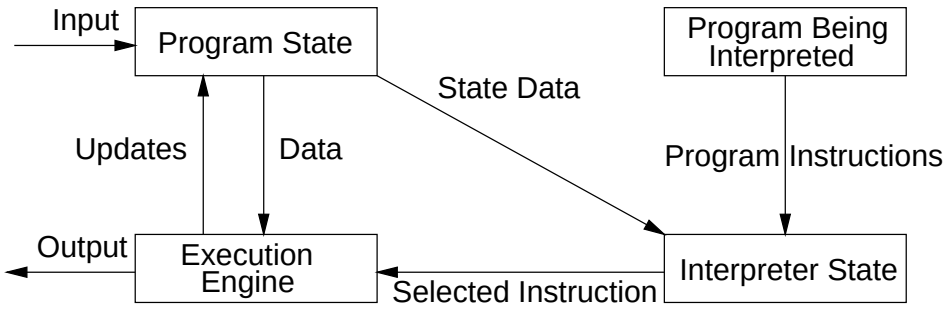

Figure 6.6: Principal Schema of Virtual Machines

Examples: Interpreters such as the Java virtual machine, syntactic shells and expert systems systems that perform by executing of production rules.

Significant Variants: Since expert systems require complex rule selection procedures various specialized forms have evolved for this application domain.

The elements of the reference model of (unbundled) ADBMSs elaborated in Section 6.1 are mapped as follows to the virtual machine architecture style: The *rulebase* contains the counterpart of the program to be interpreted, whereby the rules act as program statements. The *rule execution engine* corresponds to the interpretation engine depicted in Figure 6.6. Hence, the *scheduler* is conceived as an interpreter state component selecting the next rule to be executed. Event occurrences and transaction states

(i.e., which transactions have been started and whether and in which way they have terminated) as well as eventual rule assertion points form the counterpart of state data in Figure 6.6. The input of this virtual machine is formed by the event signals (including rule assertion points), whereas the DBMS commands (queries and DML statements) signal that a transaction may resume when all event occurrences are processed. Invocations of external programs (including database manipulations) are the output of the virtual machine.

The virtual machine style covers many aspects of an ECA System, but it is not completely sufficient as reference architecture of an ADBMS. A typical virtual machine is principally outlined to execute a program which is loaded once into the interpreter and is not modified at runtime. Instead the rulebase of an ADBMS is intended to be modified over the lifetime of an active database. As a consequence the rulebase delivers rule definitions to the scheduler in order to build the rule instantiations. Furthermore, the rulebase invokes the event service in order to subscribe respectively unsubscribe to event definitions and the rule execution component to compile action and condition definitions which are executed subsequently in the passive DBMS or in external clients. Finally an event service not only receives event signals as input but it also sends output to external event sources in order to subscribe for specific events.

Thus the reference architecture of an ECAS represents a modified virtual machine as shown in Figure 6.7. Subsequently the components depicted in Figure 6.7 are re-

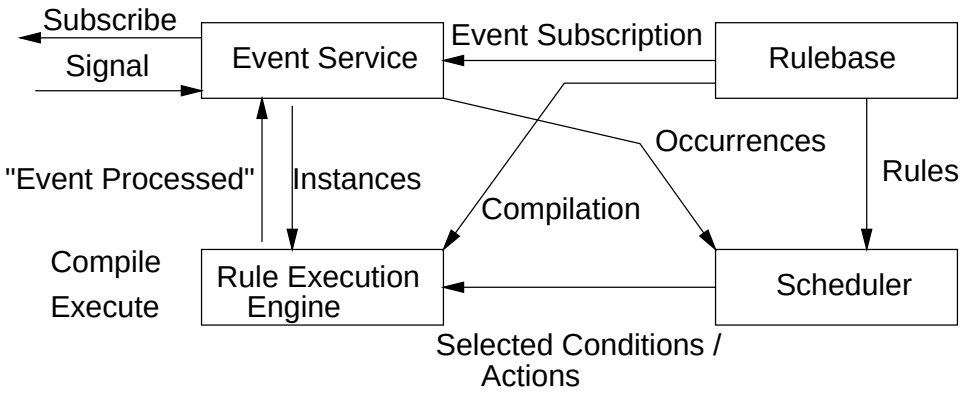

Figure 6.7: ECA System as Virtual Machine

ferred to as *major components*.

Hierarchical Decompositions in WRIGHT

Components and connectors that are composed of architectural subsystems are defined in WRIGHT as *hierarchical* descriptions. Thus the computation of a component or the glue of a connector is represented as a configuration. In addition, however, for a component the nested description has an associated set of bindings, which define how the unattached port names from internal components are associated with the port

names of the embracing component (correspondingly for connectors: role names on the inside are identified with the role names on the outside).

In order to specify the decomposition of an ECAS into subcomponents, we define a computation as exemplified in the following fragment.

Computation ecasAsVM(nEvSrc: 1 . . . ; nEvCh: 1 . . . ; nRuleCh: 1 . . . ;
nRbClients: 1. . . ;) =
Configuration VirtualMachine
. . .
Component SimpleEventService(nEvSrc: 1 . . . ; nEvCh: 1 . . . ;
C: Computation)
Port $\text{EvDef}_{1...nEvSrc}$ = IClientPull
Port $\text{EvIn}_{1...nEvCh}$ = IServerPush
Port EventSubscr = IServerPush
Port . . .
Computation = C
Component SimpleRulebase . . .
Connector ClientServer . . .
. . .
Instances
E: SimpleEventService(. . .)
CS1: ClientServer
R: SimpleRulebase
. . .
Attachements
E.EventSubscr **as** CS1.Server
. . .
End VirtualMachine
Bindings
$\text{E.EvDef}_{1...nEvSrc}$ **as** $\text{EvDef}_{1...nEvSrc}$
$\text{E.EvChnl}_{1...nEvChn}$ **as** $\text{EvChnl}_{1...nEvChn}$
. . .
End Bindings
End VirtualMachine
End ecasAsVM

The bindings `EvDef` and `EvChnl` refer to the ports of the embracing ECAS specification of Figure 6.4. It is principally sufficient to replace in the configuration in Figure 6.5 the definition of the computation `MonolythicECAS` with that of `ecasAsVM` to achieve a consistent specification of a decomposed ECAS.

6.2.3 Subsystem Design

Conceiving an ECAS as virtual machine as shown in Figure 6.7 gives a basic structure of the architecture and can therefore be used as "system metaphor" of an ECAS according to the Extreme Programming methodology (cf. Sec. 5.2.1). In this section the major components are specified architecturally and refined into subcomponents – as far as *strategic decisions*[6] are concerned. Tactical decisions are made in Chapter 7 when component provision takes place.

This section is organized as follows: First the role of the rulebase in the reference architecture is clarified. Subsequently the event service and the scheduler are decomposed into their major subcomponents. Finally, the organization rule execution component is established.

The Rulebase

As seen in Section 6.2.2 a rulebase serves as a central storage that collaborates with all major components of an ECAS. Such a behaviour is typically designed according to the so-called *repository architecture style*:

Problem: Applications in which the central issue is establishing, augmenting and maintaining a complex central body of information. Typically the information must be manipulated in a wide variety of ways.

Context: Repositories often require considerable support, either from an augmented runtime system (such as a database) or a generator to process the data definitions.

Solution:

- *System Model:* centralized data, usually richly structured.
- *Components:*
  - One so-called *blackboard* which is a central data structure representing the current state of the system.
  - Many purely computational processes that operate on the blackboard.
- *Connectors:* Computational units interact with the memory by direct data access or procedure calls.
- *Control Structure:* Varies with type of repository; may be external (depends on input data stream for databases), or predetermined (as for compilers) or internal (depends on state of computation as for active repositories.

Diagram: According to Figure 6.8

[6] In [Boo94] *strategic decisions* are defined as those which have sweeping architectural implications, whereas *tactical decisions* have only local architectural implications.

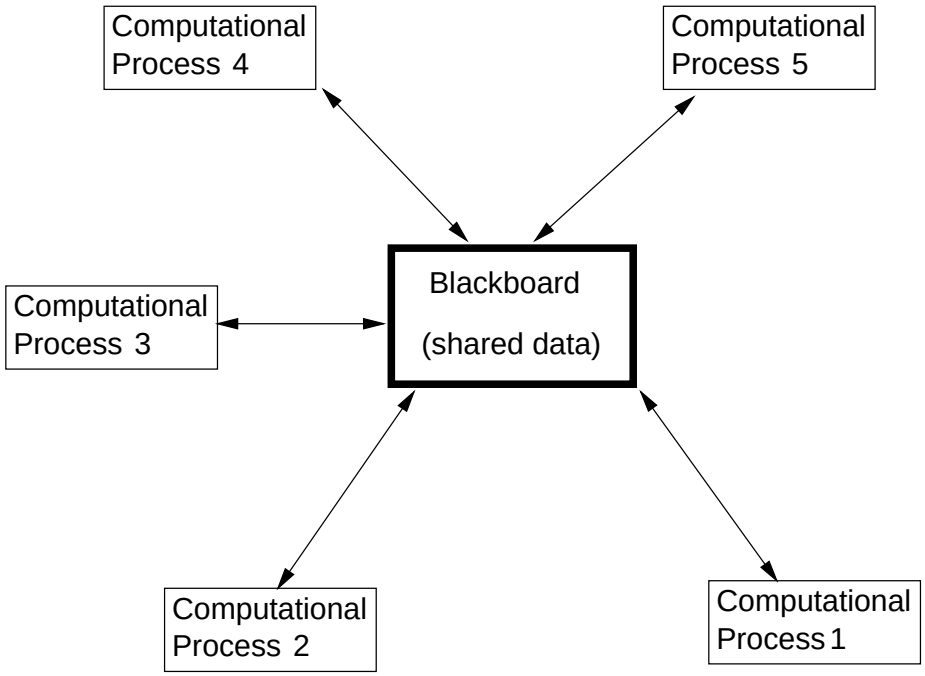

Figure 6.8: Principal Schema of the Repository Architectural Pattern

Significant Variants: The repository pattern covers large centralized transaction-oriented databases as well as blackboard systems applied for some AI applications and systems with predetermined execution patterns in which different phases add information to a single complex data structure. These variants differ chiefly in their control structure. Often long-term persistence may also be required. It is feasible to design the blackboard as an active repository that is connected to the observing components by implicit invocation. In that case, the blackboard sends notifications to the subscribing components when data of interest changes, whereby the notified components can query the blackboard in turn to get more detailed information.

Thus we can specify the rulebase as follows:

Component Rulebase(nRbAcc: 1...)
Port $\text{RbAcc}_{1...nRbAcc+1}$ = IServerPush
Port EventSubscr = IClientPull
Port CondActLib = IClientPull
Computation =

$$\Big\| \forall\, i \in nRbAcc + 1 \Big\| \; RbAcc_i.open \rightarrow Listen_i$$

where $\text{Listen}_i = RbAcc_i.request?x \rightarrow (\overline{RbAcc_i.result!y} \sqcap$
$\overline{EventSubscr.open} \rightarrow \overline{EventSubscr.request!event} \rightarrow$
$EventSubscr.result?x \rightarrow \overline{EventSubscr.close} \rightarrow \overline{RbAcc_i.result!y}$

$$\sqcap\ \overline{CondActLib.open} \rightarrow \overline{CondActLib.request!event} \rightarrow CondActLib.result?x \rightarrow \overline{CondActLib.close} \rightarrow \overline{RbAcc_i.result!y}$$

The `RbAcc` ports are attached to their counterparts of the embracing ECAS. There is an additional `RbAcc` port for the scheduler to query the rulebase in order to create the rule instantiations. Through the port `EventSubscr` the rulebase invokes the event service in order to subscribe for event definitions. Correspondingly the rule executor is invoked via the `CondActLib` port to create executable conditions and actions.

The Event Service

The event service basically records (primitive) event signals as event instances, detects eventually occurring composite events and maintains a (persistent) event history. These tasks are performed in a well-defined order whereas complex event detection is not a mandatory element of an event service, as there can be rule models without complex events. Thus the *pipes and filters architecture style* is adequate to design the architecture of the event service.

Problem: The pipes and filters style is suitable for applications that require a series of independent computations to be performed on ordered data. It is particularly useful if each of the computations can be performed incrementally on a data stream. In such cases the computations can principally proceed in parallel.

Context: The style relies on being able to decompose the problem into a set of computations (*filters*) that transform one or more input streams *incrementally* to one or more output streams. Filters must be independent entities and must not share state with other filters to perform the computation. Hence, filters are not required to know the identity of their upstream and downstream filters.

Solution:

- *System Model:* data flow between components, with components that incrementally map data streams to data streams.
- *Components:* filters which are purely computational, local processing and asynchronous.
- *Connectors:* data streams.
- *Control Structure:* data flow.

Diagram: According to Figure 6.9

Significant Variants: It is feasible to extend the pipes and filters style by restricting what appears on the input pipes of a filter or to make guarantees about what

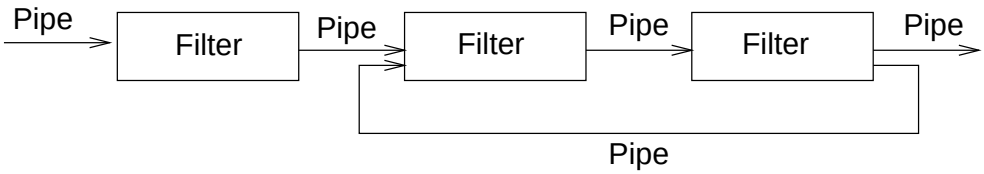

Figure 6.9: Pipes and Filters

appears on its output pipes (i.e, *typed pipes*) still without identifying the components at the end of those pipes. Furthermore it is not uncommon that the filters obtain control information from components that do not reside in the pipeline. If such modifications are applied one speaks about a so-called *modified* pipes and filters style [GS94] which is in fact the typical application of this style.

Writing to a pipe is defined in WRIGHT as follows:

Interface Type IWritePipe = $(\overline{open} \rightarrow WriteOutput) \sqcap \S$
where WriteOutput = $(\overline{write!x} \rightarrow WriteOutput) \sqcap (\overline{close} \rightarrow \S)$

Correspondingly reading to a pipe is defined by the `DataInput` interface

Interface Type IReadPipe = $(\overline{open} \rightarrow ReadInput) \sqcap \S$
where ReadInput = $(\overline{read} \rightarrow (data?x \rightarrow ReadInput) \Box eof \rightarrow \overline{close} \rightarrow \S))$
$\sqcap (\overline{close} \rightarrow \S)$

The connector linking two filters – a so called *pipe* – must deliver the data received on the reading end in the same order on the writing end of the pipe. Furthermore it must ensure that no data is lost when one of the two filters disconnects from the pipe. The following WRIGHT specification complies to these requirements:

Connector Pipe =
Role Source = IWritePipe
Role Sink = IReadPipe
Glue = $Open_{\langle\rangle}$ **where**

$$Open_s = \begin{cases} Source.write?x \rightarrow Open_{\langle x \rangle} & \\ \Box\ Source.close \rightarrow Closed_{\langle\rangle} & \\ \Box\ Sink.close \rightarrow Capped & \textbf{when} \quad s = \langle\rangle \\ & \\ Sink.read \rightarrow \overline{Sink.data!x} \rightarrow Open_{s'} & \\ \Box\ Source.write?y \rightarrow Open_{\langle y \rangle \frown s' \frown \langle x \rangle} & \\ \Box\ Source.close \rightarrow Closed_{s' \frown \langle x \rangle} & \\ \Box\ Sink.close \rightarrow Capped & \textbf{when} \quad s = s' \frown \langle x \rangle \end{cases}$$

$$Closed_s = \begin{cases} Sink.read \rightarrow \overline{Sink.data!x} \rightarrow Closed_{s'} & \\ \quad \square\ Sink.close \rightarrow \S & \textbf{when} \quad s = s' \frown \langle x \rangle \\ Sink.read \rightarrow \overline{Sink.end-of-data} \rightarrow Sink.close \rightarrow \S & \\ \quad \square\ Sink.close \rightarrow \S & \textbf{when} \quad s = \langle\rangle \end{cases}$$

$$Capped = Source.write?x \rightarrow Capped$$
$$\quad \square\ Source.close \rightarrow \S$$

An event service must provide on the one hand ports that are bound to `EventDef` and `EventChn` ports of the embracing ECAS as well as a port that enables the rulebase to invoke event subscription. Hence these ports are identical to their counterparts in the ECAS and the rulebase respectively. On the other hand there are several novel ECAS-internal communications for which adequate ports must be specified (cf. Fig. 6.7):

- The event service forwards event occurrences to the scheduler via the port `EvOcc`. Since the event service principally receives the event signals in an external format and transforms them into event occurrences which are suited for subsequent rule processing, the event service and the scheduler are conceived as filters in a pipeline.
- The rulebase invokes event subscription through the ports `EvSubscr`. For each event type in the rule model of the ECAS one such port is defined.
- Through `EvState` event instances are retrieved by other components.
- The event service is informed by `EvProc` that all rules for a specific event have been processed.

Thus the interface of the event service is defined as follows:

Component EventService(nEvTypes: 1 ...; nEvSrc: 1 ...; nEvCh: 1 ...: C: Computation)
Port $\text{EvDef}_{1...nEvSrc}$ = IClientPull
Port $\text{EvIn}_{1...nEvSCh}$ = IServerPush
Port EvOcc = IWritePipe
Port $\text{EvSubscr}_1 \ldots nEvTypes$ = IServerPush
Port EvState = IServerPush
Port EvProc = IServerPush
Computation = C

The event service is decomposed into a pipeline as shown in Figure 6.10. The pipeline starts with an *event signal processor*, that multiplexes primitive event occurrences from various event sources and forwards them to the next filter. The interface of the

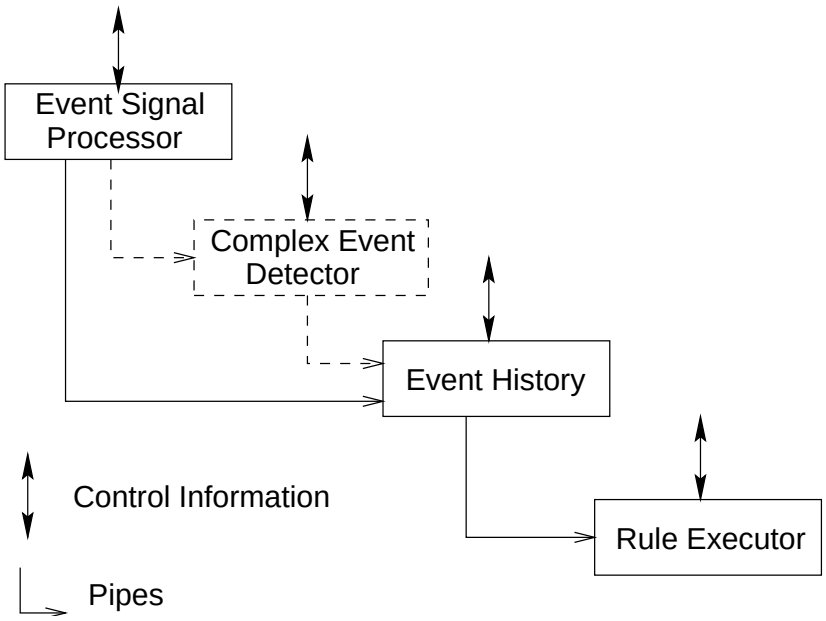

Figure 6.10: The Reference Architecture of the Event Service

WRIGHT specification of the event signal processor component corresponds to a large extent to the interface specification of the event service. There are the identical ports `EvDef`, `EvIn`, `EvSubscr` and `EvProc` which are bound to their counterparts of the event service component. The port `EvOcc` is the writing end of the filter, however it is not bound to the respective port of the event service but is connected via a pipe to the next subcomponent of the event service .

The complete WRIGHT specification of the event signal processor is as follows:

Component EventSignalProc(nEvTypes: 1 . . . ; nEvSrc: 1 . . . ; nEvCh: 1 . . . :
C: Computation)
Port $\text{EvDef}_{1...nEvSrc}$ = IClientPull
Port $\text{EvIn}_{1...nEvCh}$ = IServerPush
Port EvOcc = IWritePipe
Port $\text{EvSubscr}_1 \ldots nEvTypes$ = IServerPush
Port EvProc = IServerPush
Computation = $EvProc.open \rightarrow \forall\, i \in nEvSrc\overline{EvDef_i.open} \rightarrow$
$\overline{EvOcc.open} \rightarrow Operate \;\square\; \S$
where Operate =
$\Big\| \forall\, i \in nEvSrc \Big\| EventChnl_i.open \rightarrow Listen_i$
where $\text{Listen}_i = EvIn_i.request?event \rightarrow \overline{EvOcc.write!event} \rightarrow Listen_i$
$\square\; EvIn_i.close \rightarrow \S$
$\| EvProc.close \rightarrow \forall\, i \in nEvSrc\overline{EvDef_i.close}$
$\| \forall\, i \in nEvTypesEvSubscr_i.open \rightarrow Subscribe_i$

where Subscribe$_i$ = $\overline{EvSubscr_i.request?eventDef} \rightarrow$
$\exists j \in 1 \ldots nEvSrc \bullet \overline{EvDef_j.request!eventDef} \rightarrow EvDef_j.reply?x \rightarrow$
$\overline{EvSubscr_i.reply!x}$
$\Box\ EvSubscr_i.close \rightarrow \S$

Note that the event signal processor incorporates the various DBMS- and operating system-specific event detection adapters. Since the specification of these adapters is not a strategic issue we omit their specification in the reference architecture but consider them at the component provision level (cf. Chap. 7).

The complex event detector and the event history can be conceived at the architecture level as simple modified filters which are specified in WRIGHT as follows:

Component ModifiedFilter
Port Inp = IReadPipe
Port Outp = IWritePipe
Port Ctrl = IPushServer
Computation = $CtrlInfo.open \rightarrow \overline{Inp.open} \rightarrow \overline{Outp.open} \rightarrow Operate$
where Operate = $\overline{Inp.read} \rightarrow$
$(Inp.data?x \rightarrow \overline{Outp.write!x} \rightarrow Operate \Box eof \rightarrow$
$\overline{Outp.close} \rightarrow Capped)$
$\Box\ Ctrl.close \rightarrow\rightarrow \overline{Inp.closeOutp.close} \rightarrow \S$
where Capped = $Ctrl.request?x \rightarrow \overline{Ctrl.reply!y} \Box Ctrl.close \rightarrow \S$

The following computation stands for an event service that provides composite event detection.

Computation EventServiceComp(nEvSrc: 1 . . . ; nEvCh: 1 . . . ; nRuleCh: 1 . . . ;
nRbClients: 1. . . ;) =
Configuration ComplexEvents
. . .
Component EventSignalProc(nEvTypes: 1 . . . ; nEvSrc: 1 . . . ; nEvCh: 1 . . . :
C: Computation)
Component ModifiedFilter . . .
Connector Pipe . . .
Connector ClientServer . . .
. . .
Instances
SignalProc: EventSignalProc(. . .)
ComplexEvDet, EvHistory: ModifiedFilter
pipe1, pipe2: Pipe
. . .
Attachements

SignalProc.EvOcc **as** pipe1.Source
ComplexEvDet.Inp **as** pipe1.Sink
ComplexEvDet.Outp **as** pipe2.Source
EvHistory.Inp **as** pipe2.Sink
...
End Attachements
Bindings
SignalProc.EvDef$_{1...nEvSrc-1}$ **as** EvSubscr$_{1...nEvSrc-1}$
SignalProc.EvChnl$_{1...nEvChn}$ **as** EvChnl$_{1...nEvChn}$
SignalProc. **as** EvProc
ComplexEvDet.Ctrl **as** EvSubscr$_{1...nEvSrc}$
EvHistory.Ctrl **as** EvState
EvHistory.Outp **as** EvOcc
End Bindings
End ComplexEvents
End EventServiceComp

The loose coupling of the components in pipes and filter architecture style enable an elegant design for event services that provide complex event detection as an optional feature. In the most primitive form of an event service, the next filter is the event history where the event instances are registered. The event history subsequently forwards the registered event instances to the scheduler. Should the rule model of an ECAS incorporate complex event detection, it is feasible to simply put a complex event detector between the event signal processor and the event history.

The Scheduler

The scheduler manages the rule execution state which is defined by the set of currently triggered rule instantiations. It receives the event occurrences, determines the rules to be executed on account of these event occurrences and invokes query evaluation and action execution in accordance with the conflict resolution and cycle policy. The interface of a scheduler consists of three port categories: one port to read event occurrences, another to query the rulebase in order to create the rule instantiations and several identical ports to enable concurrent condition evaluation and action execution. From this arises the following interface definition of a scheduler:

Component Scheduler(nRuleChnl: 1...; C: Computation)
Port EvSignal = IReadPipe
Port Rulebase = IClientPull
Port RuleChnl$_{1...nRuleChnl}$ = IClientPull
Computation = C

The rule execution cycle manager is divided into two subcomponents that are related in the modified pipe and filter architecture as shown in Figure 6.11.

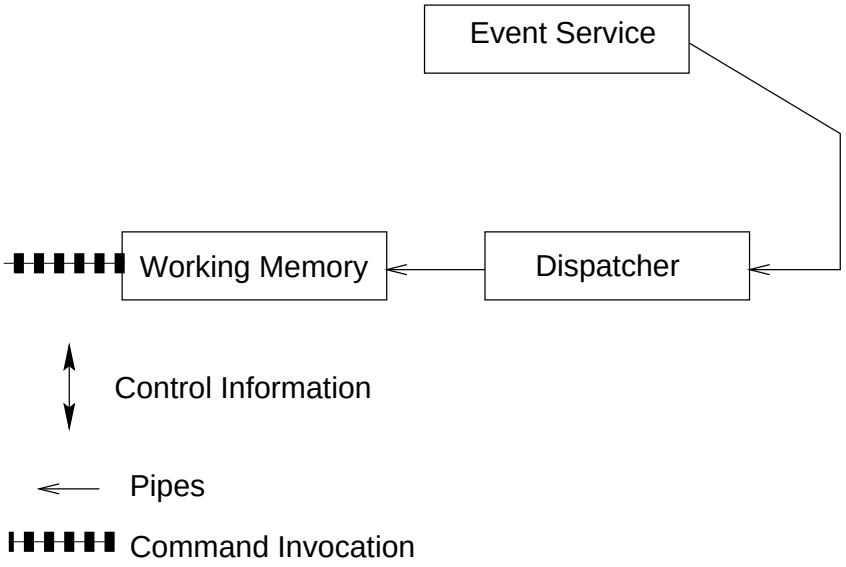

Figure 6.11: The Reference Architecture of the Rule Execution Cycle Manager

1. A *dispatcher* that receives the signalled events and generates so-called *rule instantiations* for each rule triggered by the respective event occurrence. The rule instantiations record status information of a triggered rule such as the event parameters, whether the condition has already been evaluated etc. They are subsequently forwarded to the outgoing pipe.

2. The *working memory* manages all rule instances that are currently executed and invokes condition evaluation and action execution respectively. The order of rule processing is determined according to rule priorities and the respective conflict resolution strategy. New rule instances are integrated into the working memory according to the respective cycle policy. Rule instances that are associated to a triggering transactions are not only kept in the working memory until their actions have been executed but until the triggering transaction ends. Thus it is feasible to perform *compensating actions*[7] upon an eventual abort of the triggering transaction.

The dispatcher is a modified filter whereby to control information is queried from the rulebase upon arrival of an event occurrence. As a consequence the specification corresponds largely to the one given above with the exception that the `Ctrl` port is of the interface type `IClientPull`. For the sake of shortness the WRIGHT specification of the dispatcher is skipped.

The working memory is the final element of the pipeline starting with the event signal processor. It invokes the rule execution engine in a “call and return” manner,

[7] Action definitions may include a compensating action

e.g., the engine evaluates a condition and returns its result. The WRIGHT specification of the working memory is as follows:

Component WorkingMem(nRuleChnl: 1...)
  Port Inp = IReadPipe
  Port Ctrl = IServerPush
  Port $\text{RuleChnl}_{1...nRuleChnl}$ = IClientPull
  Computation = $Ctrl.open \rightarrow \overline{Inp.open} \rightarrow \forall\, i \in nRuleChnl \bullet \overline{RuleChn.open} \rightarrow$
    $Operate \;\square\; \S$
    where Operate = $\Big|\Big|\; ReadInp \;\Big|\Big|\; ExecRule_j \;\Big|\Big|\; Ctrl.close \rightarrow Close$
      whereReadInp = $\overline{Inp.read} \rightarrow Inp.data?x \rightarrow ReadInp$
      where $\text{ExecRule}_j = \forall\, j \in nRuleChnl \bullet \overline{RuleChnl.request!x} \rightarrow$
        $RuleChnl.reply?y \rightarrow ExecRule_j$
      where Close = $\overline{Inp.close} \rightarrow \forall\, j \in nRuleChnl \bullet \overline{RuleChnl.close} \rightarrow \S$

Note the working memory is partitioned into so-called *rule execution threads* to enable concurrent rule execution along the `RuleChnl` ports.

The Rule Execution Engine

The rule execution engine acts at rule execution time as a server to evaluate conditions and execute actions whereby it invokes (ECAS-) external facilities to perform specific operations (e.g., in order to query the database). Thus it has an equal number of rule channel ports like the scheduler, an identical number of rule operation channels like the embracing ECAS and connections to signal that all rule instances for a specific event occurrence have been processed. Furthermore the rule execution component needs access to the event service in order to query past event occurrences.

At rule definition time the rule execution engine translates conditions and actions into an executable form. These operations are invoked via a specific *rule translation* port. In order to translate rule commands, the rule execution component relies on external services to compile proprietary operations (e.g., an action that is defined as a vendor specific stored procedure). These external facilities are invoked via the operation definition channels that are bound to their counterparts of the ECAS. Thus the WRIGHT specification has the following interface:

Component RuleExecutor(nRuleChnl: 1...; nOpDef: 1...; nOpChnl: 1...)
  Port Translate = IServerPush
  Port $\text{RuleChnl}_{1...nRuleChnl}$ = IServerPush
  Port EvState = IClientPull
  Port EvProc = IClientPull
  Port $\text{OpDefs}_{1...nOpDef}$ = IClientPull
  Port $\text{OpChnl}_{1...nOpChnl}$ = IClientPull
  Computation = ...

The rule execution engine is basically conceived as a component that coordinates these operations by providing "glue" and "veneer". The glue ties the adapters through which an ECAS outputs commands together, whereas the veneer papers over them by providing a consistent interface to all adapters, so that the scheduler does not have to deal with them individually. The layered architecture style addressed in Section 4.2.2 is adequate to design the rule execution engine as depicted in Figure 6.12. On top is the *Execution Layer* that provides the veneer by means of the Rule Execution API. It interprets the rule execution commands which are written in a specific condition/action language (cf Chap. 8) and identifies the tasks that must be performed externally. The condition/action language may vary from various ECAS (e.g., Java for one ECAS, PL/SQL [Ora92] for another, providing an application specific language is also conceivable) thus the condition/action interpreter is devised as an exchangeable component.

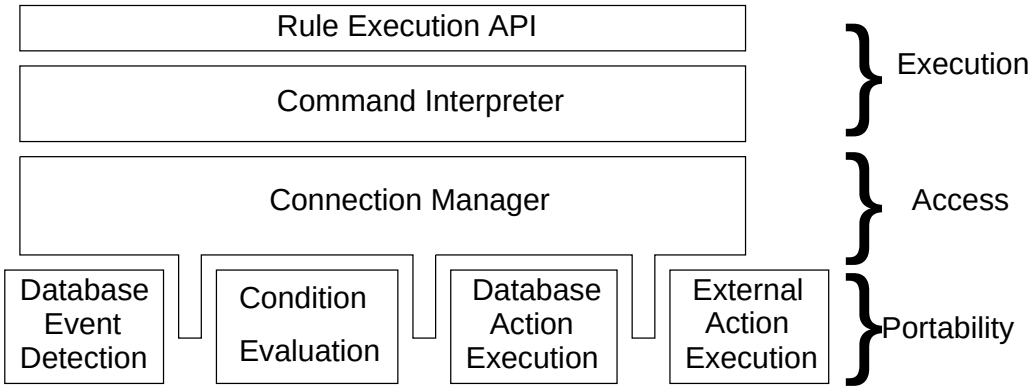

Figure 6.12: The Reference Architecture of the Rule Execution Engine

Right below the execution layer is the *Access Layer* (represented by the connection manager component) providing glue services such as routing of the command to the adequate connector, connection management to the DBMS and external processes. Furthermore the connection manager simulates specific database behavior like the establishment of a temporary database session to perform detached rule execution if a DBMS offers sibling transactions.

Finally, the *Portability Layer* encapsulates the idiosyncrasies of the respective DBMS or OS by means of various system specific adaptors.

6.3 Conclusion

In this chapter a reference model of ADBMSs has been established and transformed systematically into a reference architecture of the ECA Systems. This reference architecture is beforehand a necessity to furnish FRAMBOISE with an initial set of reusable software components which is discussed in the next chapter. However, the reference

architecture is also helpful for ADBI novices to learn how ECASs are built and supports experienced ADBIs to compare ECASs in order to interchange components from different systems. Thus the reference architecture of the ECASs is the pivotal element between the unbundling of ADBMSs and the subsequent rebundling of ECASs.

Chapter 7

Component Provision

This chapter elaborates the procedure according to which software components are developed in FRAMBOISE. It is known that no proper methodology to develop reusable software components has been found so far [Szy97]. For instance the *Catalysis* method [DW99] is devised to provide software components, but this approach does not indicate an exact development procedure. Instead it integrates different techniques into a coherent kit which must be tailored and enhanced for the respective application domain.

Thus the conception of such a method is an indispensable prerequisite to furnish a component-based construction system like FRAMBOISE. The chapter is organized as follows: Section 7.1 introduces the principles of component development, followed by a presentation of the approach conceived for FRAMBOISE (Sec. 7.2). In Section 7.3 a comprehensive example in order to illustrate the procedure is provided. Finally, Section 7.4 concludes the chapter by giving an overview of the so-called *FRAMBOISE Development Framework*.

7.1 Foundations

In this section the basic problems to implement software components are discussed in paragraph 7.1.1. Object-oriented programming techniques and in particular class frameworks are considered as the most promising approach to implement software components. Thus Section 7.1.2 introduces the basic characteristics of object-oriented class frameworks. Since class frameworks and component software appear at the first glance to be similar ideas, these concepts are compared in Section 7.1.3. Finally, the techniques to develop class frameworks are discussed in Section 7.1.4.

7.1.1 Component-Oriented Programming

The term *component-oriented programming* (COP) has been coined in [SP96] and addresses the fundamental aspects of programming components. COP is defined in

the style of typical OOP definitions by requiring the support of

- polymorphism (substitutability),
- modular encapsulation (higher level information hiding),
- late binding and loading (independent deployability),
- safety (type and module safety).

Despite of the obvious resemblance of this definition with the requirements of object-oriented and modular programming, fulfill the latter techniques only a subset of the necessities of COP. The main problem is that basically all existing design methods work only *within* a component but do not cover adequately the difficulties resulting from the complex interactions and dependencies between components.

Probably the most notorious problems to be coped with are the *asynchronity* between components, *multithreading* and *implementation inheritance across component boundaries* when object-oriented programming techniques are used [Szy97]. In greater detail they pose the following difficulties:

Asynchronity Event propagation is used by all current component infrastructures as a flexible form of component instance assembly. In principle, these facilities correspond to the implicit invocation architecture style sketched in Section 6.2 and introduce the corresponding problems, in particular when events are multicasted to several recipients. Hence the system is in an inconsistent state while multicast is in progress which might, for instance, lead to false results when a component instance queries another one by regular method invocations. Furthermore event recipients might post themselves events which must be synchronized with the pending multicasts in order to preserve correct system behaviour. Finally the set of event recipients could change while a multicast is in progress or some of the recipients might raise exceptions.

Multithreading There is a substantial increase in complexity over sequential programming when multiple sequential activities are performed concurrently over the same state space. It is exceptionally difficult to debug code that uses multiple threads and complex interlocking patterns. In principle one has to solve the same concurrency problems as in the domain of DBMSs (e.g., avoiding dirty reads, lost updates etc.). However, few general-purpose programming methods support transactions and even fewer programming languages do so that most of these mechanisms need to be programmed from scratch. Multithreading basically enables a better distribution of performance as observed by clients issuing concurrent requests. However, synchronizing concurrent threads can lead to a substantial degradation of performance. Thus the *overall* performance is typically maximized by not using threads at all and by always serving the request with the shortest expected execution time first.

Implementation Inheritance Component oriented programming basically can be performed with any programming language even though the object-oriented paradigm is considered as the most promising approach [Szy97]. However, the application of inheritance[1] leads to the so-called *Fragile Base Class Problem* (FCB) which means that a compiled class should remain stable in the presence of specific transformations of the inherited elements. The FCB problem, which is in its essence a purely object-oriented problem, is severed when implementation inheritance is applied across component boundaries. In principle it is recommended to restrict implementation inheritance to situations which are easy to grasp, i.e., to moderate numbers of classes which are all under local control and are whiteboxes. In more complex situations implementation inheritance is preferably substituted by *object composition*[2] respectively *delegation* which is a much simpler form of composition than implementation inheritance.

So far no method has been invented to implement components that takes all the above issues properly into account [Szy97]. Hence, one is basically obliged to rely on existing programming technologies implying special efforts to cope with dependencies that cross component boundaries.

7.1.2 Characteristics of Class Frameworks

Assuming object-oriented programming as the best-suited technology to implement components, one has to cope with the fragile baseclass problem discussed in the previous section. We have seen that object composition limits the fragile baseclass problem and makes software principally more flexible by enabling a configuration at runtime of the target system. However, object composition comes not for free but it is harder to understand than more static solutions. Delegation (and in a broader sense object composition) are considered to work best when used in highly stylized ways, i.e., in standard patterns [GHJV95].

Object-oriented software systems that pervasively use standard design patterns are usually designed as so-called *class frameworks*[3] which are nowadays a widely recognized technology to promote reuse in object-oriented environments [Joh97].

A class framework (CF) is defined as

[1] There are two forms of inheritance. On the one hand there is the so-called *interface inheritance* (also addressed as *subtyping*) when contracts or interfaces are inherited. On the other hand *implementation inheritance*, which is also referred to as *code inheritance* or *subclassing*, implies the inheritance of implementation fragments or code.

[2] An object performs its tasks by sending messages to other objects which are considered as parts of the first object.

[3] Class frameworks are also addressed as *(Object-Oriented) Application Frameworks*, because they are typically applied to build entire applications. Since we apply this technique in FRAMBOISE primarily to provide components, we prefer here the expression class framework or synonymously *object-oriented framework* (OOF).

> ...a set of cooperating classes that makes a reusable design for a specific class of software. A framework provides architectural guidances by partitioning the design into abstract classes and defining the responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of the framework classes [GHJV95].

Class frameworks are actually specialized class libraries that embody a generic design [CP95]. Developing an application by means of a framework consists of providing the code that implements the specifics of the particular application which is not already addressed in the general solution of the framework itself. This code, called an *ensemble* [Tal95], is typically represented as specializations of abstract classes provided by the framework. The ensemble is subsequently put together with the framework in order to provide the final solution. The parts in the framework that are open to extension and customization are termed *flexible hot spots* [Pre94].

A prominent characteristic of class frameworks is the so-called *inversion of control* which is depicted using the UML notation [BRJ98b] in Figure 7.1. In traditional

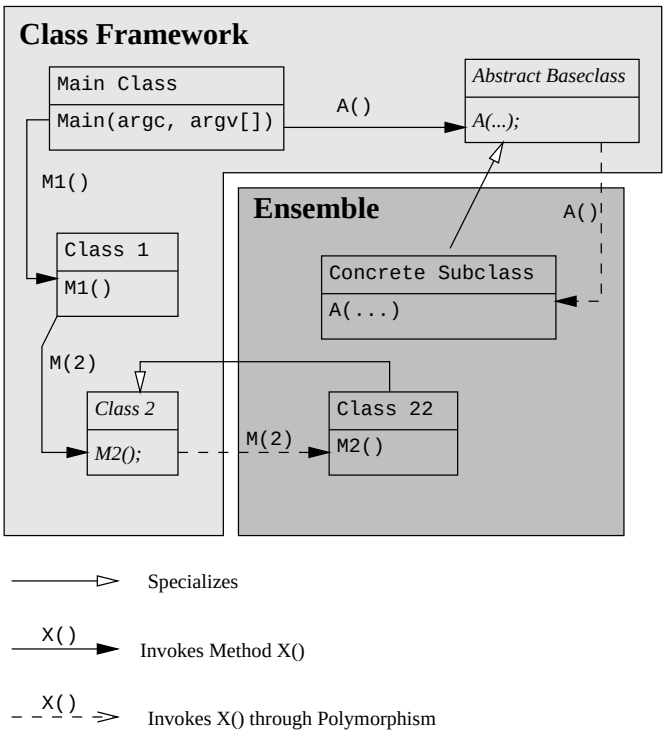

Figure 7.1: The Principal Structure of a Class Framework

programs the developer writes the main program that includes the flow of control according to which the methods of the classes that belong to the library are called. The

main program of class frameworks is, however, a part of the framework and the developer provides new classes that are "plugged" into it. Those classes are subsequently invoked by the code of the framework. This inversion of control is also known as *Hollywood Principle* (i.e., "don't call us, we call you").

Methods in a class are categorized into so-called *hook* and *template* methods [Pre94]. Hook methods are placeholders (e.g. an empty or an abstract method) that are overwritten in derived classes to be invoked by more complex methods that are implemented in the framework. These complex methods are usually termed as template methods [GHJV95]. Note that templates must not be confused with the C++ template construct which has a completely different meaning.

Class frameworks are categorized in *calling* and *called* frameworks [SBF96], where a calling framework is an active entity, proactively invoking other parts of the application whereas a called framework is a passive entity that can be invoked by other parts of the application. Note that the Hollywood Principle applies also for called frameworks, because the called framework is invoked in its entirety by the application and calls subsequently the application defined classes according to the Hollywood principle. Class frameworks are implemented software, however, they are not executable programs because they do not necessarily provide default behaviour. On account of the inversion of control, however, they are neither class libraries that are simply added to the system under construction.

7.1.3 Class Frameworks vs. Component Software

Despite of the similar names, almost identical visions and superficially similar construction principles, class frameworks differ from component frameworks [Szy97]. Component frameworks enforce, among other things, the architecture of a composite system whereas class frameworks merely structure individual components without regard of their placement in the component framework. In fact objects and class frameworks are found *within* components wherein, depending on the component's complexity, they can form their own layering and hierarchies. In contrast to the architecture of a component framework, the structure of a class framework disappears when a component is compiled. Thus a class framework is actually immaterial at runtime whereas the class instances are inexistent at compiletime.

Nevertheless, frameworks and components are considered as different but cooperating ideas [Joh97]. Besides giving the opportunity to build new components out of composite objects, class frameworks provide also the specifications for new components and a template for their implementation. Hence class frameworks are regarded as being similar to other techniques for reusing high-level design (e.g., templates [Spe88, VK89] or schemas [LH89, KRT89]) and application generators [Cle88].

> "Frameworks are firmly in the middle of reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more

> concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable) [Joh97]."

Hence class frameworks are an implementation technology that makes it easier to develop new components.

7.1.4 Framework Development Methods

Like reusable software components, class frameworks are notorious for being difficult to develop:

> "If applications are hard to design and toolkits are harder, then frameworks are the hardest of all. A framework designer gambles that one architecture will work for all applications in the domain..." [GHJV95]

Not surprisingly no fully accepted method to design class frameworks has emerged so far, but proposals to furnish application frameworks typically consist of general (but nevertheless useful) rules (e.g., [Tal95]). Most excellent object-oriented frameworks are still the product of a more or less chaotic development process, typically carried out in the realm of research like settings.

It is nowadays widely accepted that framework development should not start by trying to embrace the variability and flexibility up front. An approach to introduce flexibility stepwise into a framework is proposed in [Pre97] whereas [Sch97] suggests a method to generalize class frameworks systematically.

Even though these methods promote a sound framework development process, is it not feasible to construct an immutable (industrial-strength) class framework after a limited number of iterations [CHSV97]. Instead frameworks should be designed such that they can evolve and be easily evolved and adapted also when they are operational (i.e., used for actual application development). In order to avoid that architectural drifts and version proliferation of the framework hamper the maintenance of applications developed so far, [CHSV97] suggests the application of so-called *reuse contracts* [SLMH96] to promote cooperation between framework and application engineers.

7.2 Component Development in FRAMBOISE

In order to enable an effective component development we need basically three elements: Proper descriptions at various abstraction levels, a systematic procedure to generalize components (cf. in the component engineering section) and finally a process to develop the components effectively. Each of these issues is addressed subsequently.

7.2.1 Component Abstractions

Components are considered in the FRAMBOISE component model (cf. Sec. 4.1.5) chiefly as *abstract* components which implies the paramount importance of proper

component abstractions. They must enable a developer to consider components at various abstraction levels in order to transfer the generic vision of the reference architecture into implemented software.

Component design is performed in FRAMBOISE at three levels of abstraction:

Architectural level. The highest level of abstraction considers components from the point of view of the gross structure of the ECASs. Components are regarded as opaque entities that result from the design of the reference architecture (cf. Chap 6). Special emphasis is given to the principal interaction between components. Components and their interoperations are described by means of the architecture definition language WRIGHT which is detailed in Appendix A.

Functional level. The next lower level of abstraction assumes components as black boxes whereas their interfaces are detailed in a way that they can be mapped easily to object-oriented programming languages. In order to specify components at the functional level we apply a specification language proposed in [BBK+97]. This approach below is introduced.

Implementation level. The lowest level of abstraction copes with the actual implementation of the components. Hence the components are regarded from the point of view of the chosen programming method (i.e., object-oriented programming in our case). Aspects of the component infrastructure are equally taken into account. In order to specify components at the implementation level, we adopted the widely known Unified Modeling Language (UML). Since UML gained such a widespread popularity, this issue is not detailed in this thesis and the reader is referred to [BRJ98a].

The interrelationship among these levels of abstraction is illustrated in Figure 7.2.

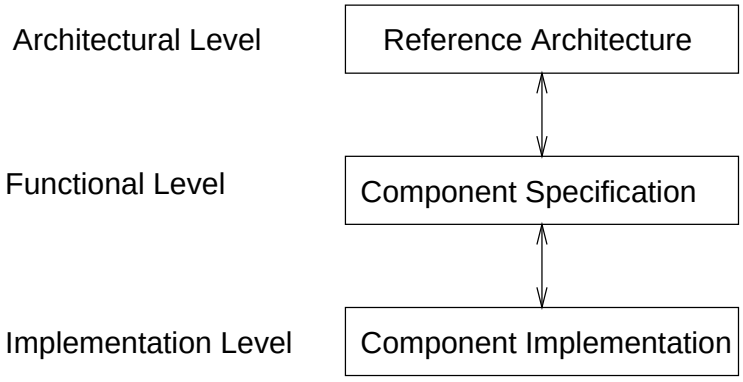

Figure 7.2: The three Levels of Component Design

The functional abstraction level was introduced basically in order to enable a smooth transition from the component description applied in WRIGHT to a chiefly object-oriented view adopted in the implementation level. In particular a clear distinction

between component interfaces and class interfaces is required in order to avoid that these concepts are blurred (cf. Sec. 4.1.2).

The method chosen to specify components at the functional level stems from an approach suggested in [BBK+97] where components are chiefly defined by concise interface definitions. Thereby interfaces are defined by constructs that are strict extensions of the CORBA IDL interface [Obj97]. Hence interface definitions consist of features such as operations, attributes, multiple inheritance and name spaces. Furthermore the interface definitions have additional semantic-oriented features of *pre-* and *postconditions*, *invariants* and *states*.

Note that this specification language has actually been devised as an architecture specification language and has therefore been named as Architecture Specification Language (ASL). We prefer WRIGHT to specify software architectures because of its focus on the interaction between components. Furthermore WRIGHT obliges a designer to abstract more effectively from implementation issues whereas the concreteness of ASL might distract a developer to "think" in terms of the implementation already at the architecture level of design.

7.2.2 Systematic Component Generalization

Component generalization is principally the only activity where component engineering differs from traditional software development (cf. Sec. 5.1.1). There is, however, no ready to use method to generalize components. Even though the principal generalization techniques (narrowing, widening, isolation and configurability; cf. Sec. 5.1.1) are known, it is not clear when and according to which design rules they shall be applied. Thus for FRAMBOISE procedures were investigated that enable a systematic component generalization. Thereby we experienced that methods to generalize class frameworks systematically are to some extent applicable to the component level.

In the next paragraph techniques enabling a systematic class framework generalization are discussed, followed by a description of how these techniques are applied in FRAMBOISE.

Systematic Framework Generalization

A method to generalize class frameworks systematically is proposed in [Sch97]. This procedure separates the development of a class framework into several distinct activities:

1. A class model for a fixed application is designed.
2. In a *hot spot high-level analysis* all hot spots are collected and briefly described.
3. The *hot spot detail analysis and specification* activity is done for each hot spot. The variability and flexibility requirements are analyzed in detail and the hot spot characteristics are described according to the following list:

- Hot spot *name*.
- Short *description*.
- *Common responsibility R* that generalizes the different alternatives.
- Different *alternatives* realizing R.
- *Kind of variability* required (e.g., different kinds of bank accounts).
- Whether some or all of the variability may be covered by *parameterization*.
- *Granularity*, i.e., whether the hot spot covers one (*elementary* hot spot) or multiple variable aspects (*nonelementary* hot spot).
- *Multiplicity* gives the number of alternatives (either one or n) that are simultaneously bound to a hot spot.
- *Binding time* is either application creation or at runtime whereas the latter is distinguished between once or multiple times.

4. The *hot spot subsystem high-level design* activity derives the classes and structure of a so-called hot spot subsystem from the hot spot characteristics. Hot spot subsystems consisting of a (usually abstract) base class, concrete derived classes representing the alternatives and eventually further helper classes and relationships. Design patterns [GHJV95] help to determine the detail structure of a hot spot by describing typical, common and frequently observed relationships among classes.

5. A *generalization transformation* introduces the variability and flexibility of a hot spot into the class structure. Fixed specialized class and possibly also directly related classes are replaced with a hot spot subsystem resulting from the high-level hot-spot subsystem design.

These activities are not performed in one development cycle but are distributed over several iterations. Typically an application class structure is designed and partially implemented in the initial development cycles. Henceforth a rough hot spot analysis is performed, followed by a hot spot detail analysis, class structure generalization and implementation for one or a few hot spots at a time in further development cycles. Finally *restructuring cycles* to refactor aspects that were not well understood are considered as an intrinsic part of the framework generalization process.

Component Generalization in FRAMBOISE

Component generalization takes place in FRAMBOISE at all component abstraction levels. At the architecture and the implementation level a general applicability of the software components is fostered by the adoption of the architecture styles and the techniques to develop class frameworks respectively. As far as is known no method to generalize components at the functional level has been proposed so far.

Such a procedure is, however, necessary even though the architecture and framework design techniques provide for some generality of the software components. Recall that designing a modifiable reference architecture and implementing it straightforwardly as a class framework – thereby hoping for the best that the framework implementation techniques achieve a reasonable component generalization – implies that the notion of a component and that of a class are blurred in one of the most critical phases of the unbundling process. Such a "systematic confusion" is an unsound and error-prone engineering practice which yields usually bad designs (e.g., which generalized class shall be considered subsequently as component?).

It is, however, feasible to adopt some elements of the framework generalization techniques described above to generalize functional descriptions of components. The hot spot analysis and specification are essentially independent from the respective software development technique and can therefore be used to identify the variable aspects of a component framework. The subsequent hot spot subsystem design and the generalization transformation, however, are specific for object-oriented software and are not applicable for component design.

Instead we establish some rules that enable generalization of components based on the previous hot spot analysis. These rules are directly related to the hot spot characteristics according to [FSJ99]. Hence there are the following rules:

1. *Isolation* is applied to separate components from system-specific parts like operating systems or hardware.

2. Isolation is furthermore applied if the *granularity* of a hot spot is *nonelementary*. Thus the component is split into subcomponents. Each subcomponent provides exactly one interface corresponding to one variable aspect.

3. *Configurability* is applied if the hot spot under consideration is *parameterizable*, i.e., the respective interfaces are enhanced by states that represent the parameters.

4. A component is *widened* in order to achieve uniformity if the *multiplicity* of a hot spot is *one*. Hence the variations are integrated into one single interface.

5. *Narrowing* is applied if the *multiplicity* of a hot spot is *n*. The original component is split into n subcomponents providing a specialization of the narrowed interface of the supercomponent.

These rules enable a designer to generalize functional descriptions of components in a controlled way. The benefit of this approach is that it is feasible to gain specifications of reusable software components that are *generic*, i.e., do not refer to any component infrastructure and programming method.

7.2.3 The Component Development Process

Component development is not intended to be carried out in a strict top down manner as Figure 7.2 might suggest. Instead component development is performed as a *round-trip gestalt design*, i.e., as a procedure that emphasizes the incremental and iterative development of a system through the refinement of different logical and physical views of the system as a whole. The above three levels of abstraction provide these different views so that a component developer navigates deliberately between them during the design process. As Figure 7.3 illustrates, the FRAMBOISE component development process contains the following activities:

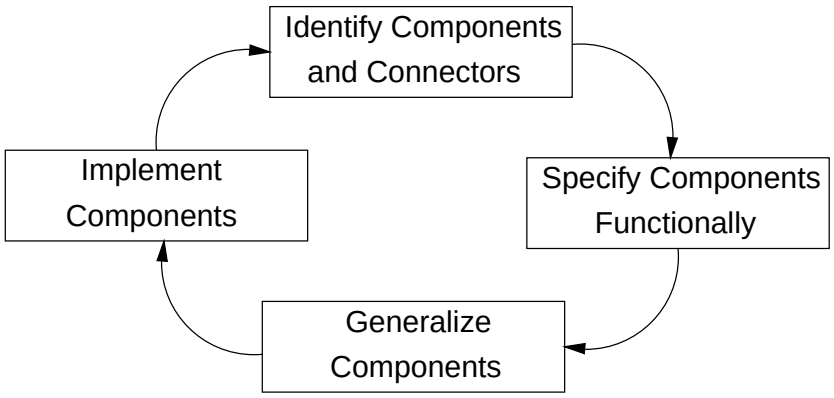

Figure 7.3: The Component Development Process

1. *Identification of the components and connectors*. The developer identifies the major components and their interconnection mechanisms in order to establish the boundaries of the problem at hand. The outcome of this activity is a description of the components at the architectural abstraction level.

2. *Functional specification of the components*. Based on the architecture specification, the developers derive detailed interface definitions (described in ASL) and establishes a first fixed-grade functional component specification.

3. *Component Generalization*. At this stage the developers perform the component generalization as described in the previous paragraph.

4. *Component Implementation*. The components are implemented by means of the chosen implementation technology and component infrastructure. The subsystem design activities to generalize the class framework structure takes also place in this phase.

Round trip gestalt design is a recognition of the fact that the big picture of a design affects its details and vice versa. Thus it is an appropriate method to perform extreme programming.

7.3 Providing the Rulebase Component

In this section we exemplify the component development process by means of a comprehensive example. The structure of this section follows the order of the component development process depicted in Figure 7.3. Hence the identification of the rulebase component is discussed in Section 7.3.1. The functional description of this component is presented subsequently in Section 7.3.2, followed by the component generalization in Section 7.3.3. Finally an approach to implement the rulebase is shown in Section 7.3.4.

7.3.1 Identification of the Rulebase Component

According to the reference architecture (cf. Sec. 6.2.2), the rulebase component maintains the rulebase persistently and provides facilities to define and modify the rulebase items[4]. Furthermore this component ensures the consistency of the rulebase, e.g., names and IDs must be unique and event definitions must not be deleted if they are part of a rule definition or a complex event definition. Finally, modifications of the rulebase are communicated to the other components of an ECAS. The rulebase is specified in WRIGHT as follows:

Component Rulebase(nRbAcc: 1...)
  Port $\text{RbAcc}_{1...nRbAcc+1}$ = IServerPush
  Port EventSubscr = IClientPull
  Port CondActLib = IClientPull
  Computation =
    $\Big\| \forall\, i \in nRbAcc + 1 \Big\| RbAcc_i.open \rightarrow Listen_i$
    where $\text{Listen}_i = RbAcc_i.request?x \rightarrow (\overline{RbAcc_i.result!y} \sqcap$
      $\overline{EventSubscr.open} \rightarrow \overline{EventSubscr.request!event} \rightarrow$
        $EventSubscr.result?x \rightarrow \overline{EventSubscr.close} \rightarrow \overline{RbAcc_i.result!y}$
      $\sqcap\ \overline{CondActLib.open} \rightarrow \overline{CondActLib.request!event} \rightarrow$
        $CondActLib.result?x \rightarrow \overline{CondActLib.close} \rightarrow \overline{RbAcc_i.result!y}$

Based on this specification it is feasible to design the functional specification of the rulebase component.

7.3.2 The Functional Component Description

In order to define the rulebase functionally, the structure of the elements to be managed by this component must be specified beforehand. We design the event, condition/action and rule definitions as first class objects according to [DPP91]. Thereby at the beginning two concrete event types are assumed, namely *abstract* and *method events*. Thus the following class structure for this rule schema is established:

[4] We summarize event condition action and rule definitions as *rulebase items*.

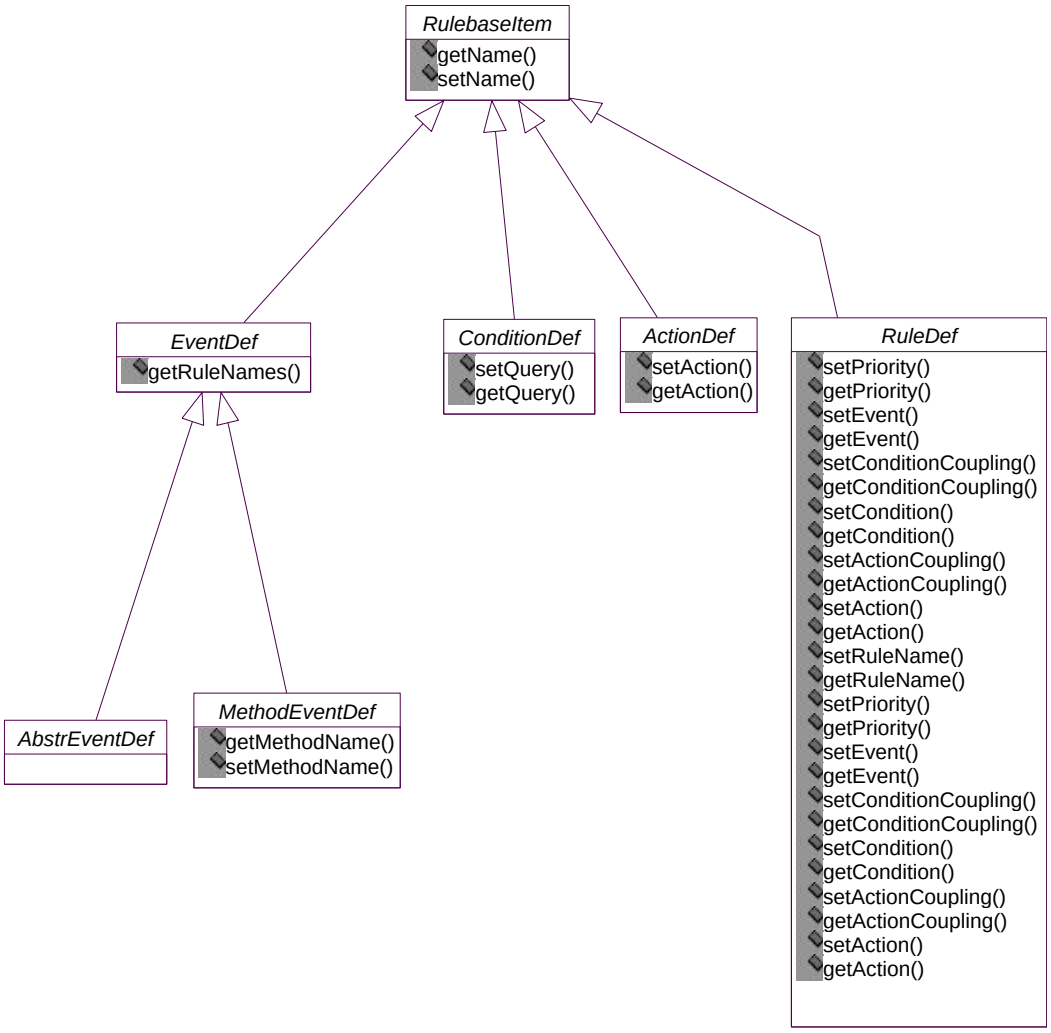

Figure 7.4: Class structure of a Simple Ruleschema

The WRIGHT component specification and the rule schema enables a basic functional component description to be established which will be generalized subsequently. From the WRIGHT description three service interfaces can be inferred forming the basic component, i.e, RbAcc implies an interface to invoke services from the rulebase. In principle the interface consists of operations to create, update, query and delete event, condition, action and rule definitions. For the sake of readability subsequently only the operations to create rulebase items are described and the other operation types are left aside. The functional description of the RbAcc interface looks as follows:

```
INTERFACE RbAcc {

  void createRulebase(in String rulebaseName)
       raises(NameClash)
//further services to open, modify and remove a rulebase

  void createAbstrEventDef(in String eventName)
       raises(NameClash)
    precondition{getEventDef(eventName) == null;}
    postcondition{getEventDef(eventName).getName(eventName)
                                          == eventName;}

  void createMethodEvent(in String eventName,
                 in String className, in String methodName,
                 in boolean beforeAfter)
    raises(NameClash)
    precondition{getEventDef(eventName) == null;}
    postcondition{getEventDef(eventName).getName(eventName)
                                          == eventName;}

//Further services to modify, delete and
//query event definitions


  void createConditionDef(in String conditionName)
    raises(NameClash)
    precondition{getConditionDef(conditionName) == null;}
    postcondition{getConditionDef(conditionName).getName()
                                           == conditionName}

  //Further services to modify, delete and
  //query condition definitions

  void createActionDef(in String actionName)
```

```
    raises(NameClash)
    precondition{getActionDef() == null;}
    postcondition{getActionDef().getName() == actionName}

//Further services to modify, delete and
//query action definitions

  void createRuleDef(in String ruleName, in int prio,
                     in EventDef event,
                     in int conditionCoupling,
                     in ConditionDef cond,
                     in int actionCoupling,
                     in ActionDef action)
    raises(NameClash)
    precondition{getRuleDef(ruleName) == null;}
    postcondition{getRuleDef(ruleName).getName()
                                        == ruleName}

//Further services to modify, delete and
//query rule definitions

}
```

The port `EventSubscr` implies an interface through which the rulebase notifies a listener (typically the event service) about modifications in the rulebase concerning event definitions.

```
interface EventSubscr {
    void RegisterEventDef(in EventDef event);

    //Further services to modify, unregister etc. events
}
```

Finally the port `CondActLib` is refined by means of an interface through which the rulebase invokes a listener (typically the rule execution component) about modifications in the rulebase concerning condition and action definitions definitions.

```
interface CondActLib {

    void CompileAction(ActionDef action);
    //Further services to modify, remove etc. actions

    void CompileCondition(ConditionDef condition);
    //Further services to modify, remove etc. conditions
}
```

An ASL component description consists of one or more *provided* and *required* interfaces as shown in the following functional description of a fixed grade rulebase component:

```
component FixedGradeRulebase {
  provides RbAcc rulebaseService;
  requires EventSubscr eventListeners;
  requires CondActLib operationLibraries;
  ...
}
```

A fragmentary event service might look as follows:

```
component BasicEventService {
    provides EventSubscr eventBase;
    ...
}
```

Composite components are specified in ASL by *binding* a required interface of one subcomponent to a provided interface of another. Thus the rulebase and event service components are integrated in an ECAS as follows:

```
component BasicECAS {
     BasicRulebase rulebase;
     BasicEventService eventService;

     bind rulebase.eventListeners
        to eventService eventBase;
}
```

The functional description of the rulebase component is a basis for the subsequent component generalization.

7.3.3 Generalizing the Rulebase Component

In order to generalize the rulebase first the hot spot analysis is performed and then each hot spot is generalized in turn.

Hot Spot Analysis

A hot spot analysis enables a generalization of the above design into the design of a reusable software component. First, the hot spot high level analysis clarifies that the following two hot spots are necessary:

1. A rulebase component must be provided in a way that different sets of rulebase items (e.g., various event types) can be managed for a different ECASs.

2. It must be feasible to build rulebase components for different persistent storage systems (e.g., DBMSs, flat files etc.).

The detailed analysis of these hot spots reveals the following characteristics:

Hot Spot 1: Variability of Rulebase Items

- Description: According to the rule model of the respective ECAS, the rulebase must be able to store the the definitions of various event types.
- Common responsibility: Provision of create, read, update and delete (CRUD) operations for the respective rulebase items.
- Examples of alternative realizations: Value, time and complex events as well as DBMS-specific database updates are different alternatives that may be supplied as event types. Furthermore condition and action definitions may be outlined for specific DBMSs or system environments.
- Kind of variability required: Maintain rulebase items from different classes.
- Parameterization is not possible.
- Granularity: elementary.
- Multiplicity: n chain structured.
- Binding time: Creation time of the ECAS.

Hot Spot 2: Variability of Rulebase Storage

- Description: A rulebase must be portable for different storage mechanisms in order to store the rulebase eventually by means of the DBMS for which an ECAS is provided.
- Common responsibility: Provides access to a persistent storage.
- Examples of alternative realizations: Flat files as well as relational or object-oriented DBMSs.
- Kind of variability required: Different storage mechanisms may be used to store a rulebase item persistently.
- Parameterization is not possible.
- Granularity: elementary.
- Multiplicity: One since a rulebase is stored entirely using the same storage mechanism.

- Binding time: Creation time of the ECAS.

Both hot spots are orthogonal, i.e., the may vary independently of each other.

Generalizing Hot Spot 1

The hot spots analyzed above provide the basis to generalize the design depicted in Figure 7.4. First, the multiplicity of hot spot 1 (Variability of Rulebase Items) is n. According to the generalization principles the rulebase component should be decomposed into subcomponents whereas we apply narrowing.

This is achieved by decomposing the rulebase component into different subcomponents – so-called *rulebase cartridges* which are responsible for a specific type of rulebase item (e.g., one specific event definition type). A rulebase cartridge provides accessor functionality to the respective rulebase item, manages it persistently and cooperation with the other rulebase cartridges. Thus the provided interface `RbAcc` must be sliced interfaces that are provided by the respective rulebase cartridge.

For example a cartridge for abstract event definitions might look as follows. First a narrowed interface is specified providing the services to query for specific event definitions.

```
interface EventAccess2Query {
  EventDef getEventDef(in String eventName)

  void deleteEventDef(in String eventName)
    precondition {(this.getEventDef() == null) or
          (this.getEventDef().getRuleNames() == null);}
    postcondition {this.getEventDef() == null;}
//Further query services that are applicable for
//arbitrary event definitions...

interface EventAccess2Modify : EventAccess2Query {
  EventDef getEventDef(in String eventName)

  void deleteEventDef(in String eventName)
    precondition {(this.getEventDef() == null) or
                (this.getEventDef().getRuleNames() == null);}
    postcondition {this.getEventDef() == null;}
//Further query services that are applicable for
//arbitrary event definitions...
```

Then we specialize this interface for an interface that enables modifiable access to generic events.

First there is its slice from the RbAcc Interface:

```
interface AbstrEventAccess : EventAccess2Modify {
  void createAbstrEventDef(in String eventName)
          raises(NameClash)
    precondition{getEventDef(eventName) == null;}
    postcondition{getEventDef(eventName).getName(eventName)
                                        == eventName;}
//Further query services that are applicable for
//arbitrary event definitions...
}
```

In order to preserve maximal flexibility, the `AbstrEventAccess` interface is accessed directly from the clients of the rulebase. Thus the abstract event cartridge must be able to query the rulebase to ensure for instance, uniqueness of the event names upon event definition.

Thus the abstract event cartridge looks as follows:

```
component AbstrEventCartridge {
      provides AbstrEventAccess;
      requires EventAccess2Query;
      requires RuleAccess2Query;
}
```

The rulebase component provides a frame where the cartridges are plugged into, mediates the interoperation between the cartridges and regulates the communication to the other components along the provided interfaces. Hence the generic rulebase component looks as follows:

```
component SimpleRulebase {

      provides EventAccess2Query;

      AbstrEventCartridge abstrEvents;
      MethodEventCartridge methodEvents;
      ConditionCartridge conditions;
      ActionCartridge actions;
      RuleCartridge rules;

   bind abstrEvents.EventAccess2Query
      to this.EventAccess2Query;

   bind abstrEvents.RuleAccess2Query
      to rules.RuleAccess;
}
```

Generalizing Hot Spot 2

According to the generalization principle 1, the hot spot 2 is solved by splitting a cartridge into a generic subcomponent that ensure general principles (such as certain consistency rules) and another subcomponent that implements the access to the respective storage facility.

The generic part is a cartridge accessor and enforces the policies that are not dependent from the chosen storage mechanism. The other subcomponent provides the persistence mechanisms. It implements the `AbstrEventAccess` interface through which the persistence facilities are accessed. Furthermore it requires an interface that gives access to specific storage facilities. For instance an abstract event definition storage component looks as follows:

```
component AbstrEv4Cloudscape {
      provides AbstrEventAccess ;
      requires DBSession;
}
```

Through the interface `DBSession` the component handles the session with the respective persistence facilities. These session handles are provided by a specific session handling subcomponent which is also a subcomponent of the rulebase component.

The generic cartridge subcomponent looks as follows:

```
component AbstrEventDefPolicy {
      provides AbstrEventAccess;
      requires AbstrEventStorage;
      requires EventAccess2Query;
      requires RuleAccess2Query;
}
```

The interface `AbstrEventStorage` specializes `AbstrEventAccess` through persistence related elements (e.g., specific exceptions etc.).

Hence a specific AbstrEventDefCartridge looks as follows:

```
component AbstrEv4Cloudscape {

      provides AbstrEventAccess;
      requires EventAccess2Query;
      requires RuleAccess2Query;
      requires DBSession;

      AbstrEventDefPolicy policy;
      AbstrEv4Cloudscape storage;

    bind this.AbstrEventAccess
```

```
      to policy.AbstrEventAccess;
   bind this.EventAccess2Query
      to policy.EventAccess2Query;
   bind this.RuleAccess2Query
      to policy.RuleAccess2Query;
   bind policy.AbstrEventStorage
      to storage.AbstrEventStorage;
   bind this.DBSession
      to storage.DBSession;
}
```

Thus the rulebase component has been generalized. Further activities to promote reusability take place when the component is implemented.

7.3.4 Implementation of the Rulebase Component

This section illustrates how the functional specification of a generalized component is mapped to the implementation for a specific component infrastructure. First the infrastructure we chose for the prototypical implementation of FRAMBOISE is presented, followed by a discussion of the principles underlying the implementation design. Subsequently the implementation of subcomponents of the rulebase (i.e., the cartridges) and the overall rulebase component is discussed.

Java, Java Beans and InfoBus

FRAMBOISE was implemented prototypically by means of the Java programming system and its component model Java Beans whereby the components were wired by means of InfoBus. Each of these elements is presented briefly in turn.

Java The Java programming language released by Sun Microsystems in the mid-1990s was designed to be hardware-independent by compiling to an artificial byte-code instruction set that could be easily implemented in an interpreter on almost any microprocessor device. Java is purely object-oriented with all data types apart from some primitive ones (integers, floats, characters etc.) as classes. The syntax is loosely based on C++ but Java eliminates many of the most difficult aspects of C++ programming. There are no pointer types in Java; all memory variables and objects are handled as references. Objects are created with a *new* operator but Java provides a built-in garbage collector that automatically deletes the objects when they go out of scope or are no longer referenced. This eliminates the memory leak problems associated with C++. Java has also the advantage of extensive class libraries.

Java Beans As Java continues to mature, a number of extensions and APIs have been added to provide additional capabilities. One is the so-called *JavaBean* standard

to create reusable software components with Java. Principally any object conforming to certain basic rules can be a bean; there is nothing like a `Bean` that all beans are required to subclass.

The main aspects of the bean model are:

- *Events.* Beans can announce (by implementing certain methods) that their instances are potential sources or listeners of specific event types.
- *Properties.* Beans expose a set of instance properties through pairs of `get...` and `set...` methods. Properties can be used for customization at assembly time as well as programmatically during execution time. Property changes can trigger (bean) events and they can be constrained to be modified only if the modification is not vetoed by specific beans (so-called vetoable listeners).
- *Introspection.* A bean can be inspected by an assembly tool to find out about the properties, events and methods that a particular bean supports.
- *Customization.* Using an assembly tool, a bean instance can be customized by setting its properties.
- *Persistence.* Customized and connected bean instances need to be saved for reloading at the time of application use.

InfoBus Normally, all beans loaded from the same classloader are visible to each other by searching the container-component hierarchy or their bean context. They can use reflection and design patterns to determine which services are provided by other beans. However, this approach is often cumbersome and prone to error. Thus Lotus Development Corporation and JavaSoft developed InfoBus as a standard approach to exchange data between beans as well as to simplify inter-bean communication.

The InfoBus operates analogously to a PC system bus. Data consumers and data producers in the same way that PC cards connect to a PC's system Bus. Data producers use the bus to send data items[5]. The InfoBus is asynchronous and symmetric. That means that producer and consumer do not have to synchronize to exchange data and any member of a bus can send data to any other member of the bus.

The InfoBus operates as follows:

- Beans, components and other objects join the InfoBus by implementing the `InfoBusMember` interface, obtaining an `InfoBus` instance and using an appropriate method to join the instance.
- Data producers implement the `InfoBusDataProducer` interface and data consumers implement the `InfoBusDataConsumer` interface. These interfaces define methods for handling events required for data exchange.

[5]The unit of data exchanged on an InfoBus is referred to as *data item* implementing the interface `DataItem`.

- Data producers signal that named data items are available on an `InfoBus` object by invoking the InfoBus's `fireItemAvailable()` method.
- Data consumers get named data items from an `InfoBus` object by invoking the `requestDataItem` method of the `InfoBusItemAvailable` event received via the `InfoBusDataConsumer` interface.

The Java component infrastructure was chosen for the prototypical implementation of FRAMBOISE because a lightweight component infrastructure such as provided by Java Beans and InfoBus was favored.

Design Principles

The rulebase implementation was designed according to the following principles.

- The rulebase component is a Java Bean. Thus components interchangeably will also be referred to as (Java) Beans.
- The rulebase is composed on subcomponents which represent also Java Beans.
- The rulebase interacts with its subcomponents by explicit invocation.
- The rulebase interacts with other components via the InfoBus.

Note that these guidelines have correspondingly been applied for the implementation of all FRAMBOISE components.

Cartridge Subcomponents

According to the functional specification, a cartridge component is decomposed into the policy and the storage subcomponent. The former provides the access to the cartridge's functionality and the operations that do not depend on the storage facilities whereas the latter stores the rulebase item persistently.

Even though the interfaces of these subcomponents are at the functional level quite similar, their bean implementations belong to different class hierarchies as shown in Figure 7.5 for the cartridge providing abstract event definitions. Since the policy subcomponent provides the public access to this cartridge, the decision was made to implement it as a data item[6] that can be invoked via the InfoBus. Storage beans, however, are not implemented as data items because they are not accessible from outside their containing cartridge. It is furthermore conceivable to specify different policy components that interoperate with the same storage components (but not with the identical component instances). The other way round, the same policy component must be able to interoperate with storage components for different persistence facilities.

[6] This name is insofar misleading as principally any object can be a `DataItem`. It is not uncommon to implement component interfaces as data items which are accessed via the InfoBus. This enables a highly dynamic deployment of Java Beans.

Figure 7.5: The Implementation of the Abstract Event Definition Cartridge

Thus, specializations of the policy and storage bean classes may vary independently, indicating that this part of the class framework should be designed according to the *bridge* design pattern [GHJV95]. In this approach the features of a hot spot are accessed via a so-called *abstraction* hierarchy that invokes methods of classes belonging to a so-called *implementation* hierarchy. Thereby the relationship between the abstraction and implementation hierarchy is provided at exactly one point, i.e., the bridge.

Note that the `EventAccess2Query` and `RuleAccess2Query` interfaces which are specified as "required" at the functional level do not appear in the bean implementation. These interfaces are merely required to enable the policy subcomponent to check for specific consistency rules (e.g., duplicate event names) in order to prevent invalid modifications at the rulebase. The bean model allows the implementation of so-called *vetoable changes* that enable specific observers to veto an operation. Thus the rulebase component might for instance veto against the deletion of an event definition that is still required in a rule. Due to the anonymity of the vetoable observers this bean-specific technique promotes an even looser coupling between components than assumed at the functional level.

The Rulebase Component

The rulebase component depicted in Figure 7.6 sets the stage to deploy the various cartridges and session handling components. It is therefore a facility of the *component framework implementation*. The rulebase ensures that all deployed subcomponents are correctly initialized and mediates afterwards the flow of information between them (e.g. vetoing the definition of an event whose name clashes with the name of a previously defined event). Thus the rulebase acts as a data controller on the InfoBus (by implementing the `InfoBusDataController` interface), i.e., as a bean that is able to control which member of the InfoBus gets what kind of events.

The basic functionality of the rulebase (e.g., creation of a new rulebase) may be accessed via the InfoBus by means of a `RulebaseAccess` data item which forwards the requests subsequently to the actual rulebase component. A scenario of an exemplary interaction of two rulebase clients with a rulebase component is depicted in Figure 7.7:

1. Client 1 requests the InfoBus for the `RulebaseAccess` item.
2. The InfoBus forwards this request as a `dataItemRequest` to the rulebase.
3. The rulebase returns the `RulebaseAccess` item to the InfoBus.
4. The `RulebaseAccess` item is subsequently handed to client 1.
5. Client 1 requests accessing rulebase "xxx" by invoking `open` on `RulebaseAccess`.

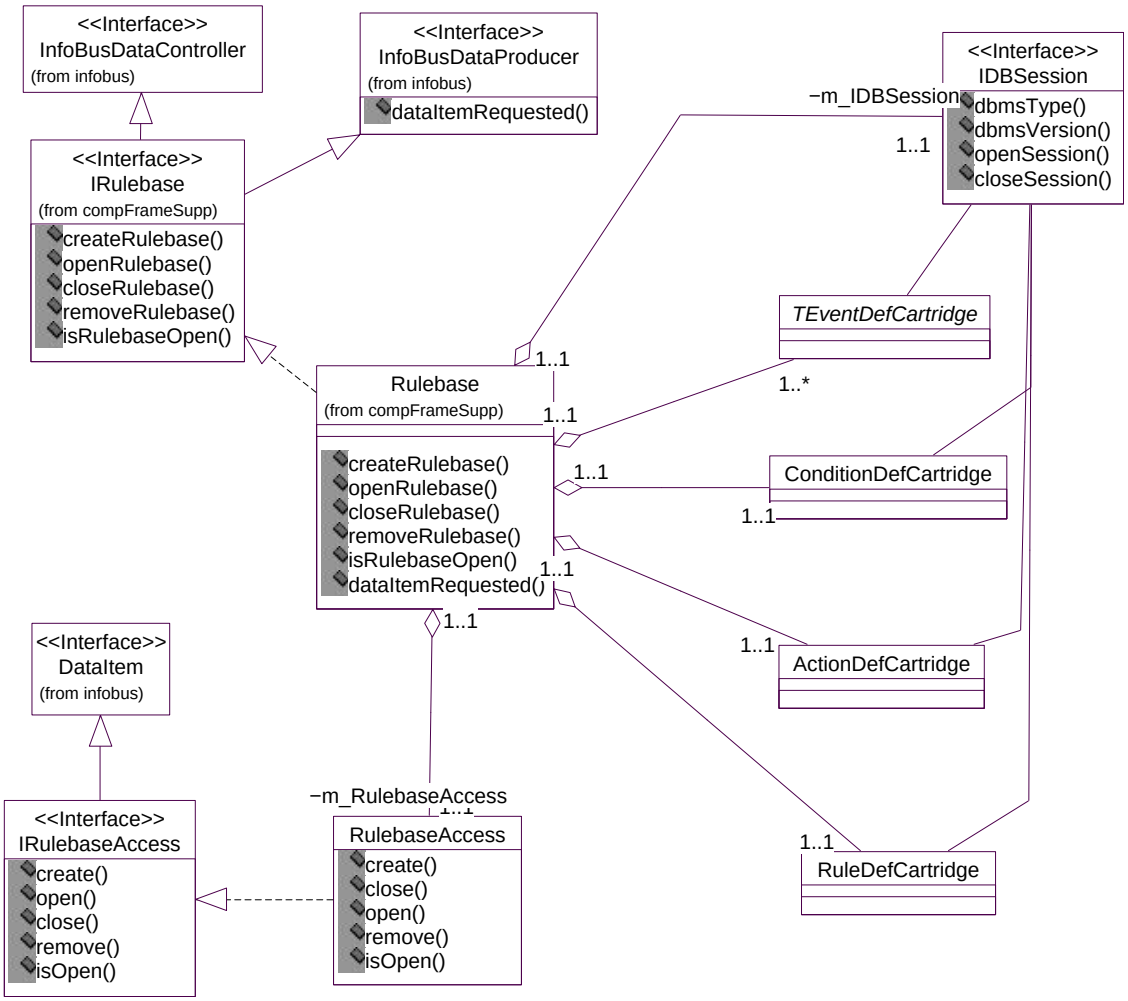

Figure 7.6: Class Design of the Rulebase Component

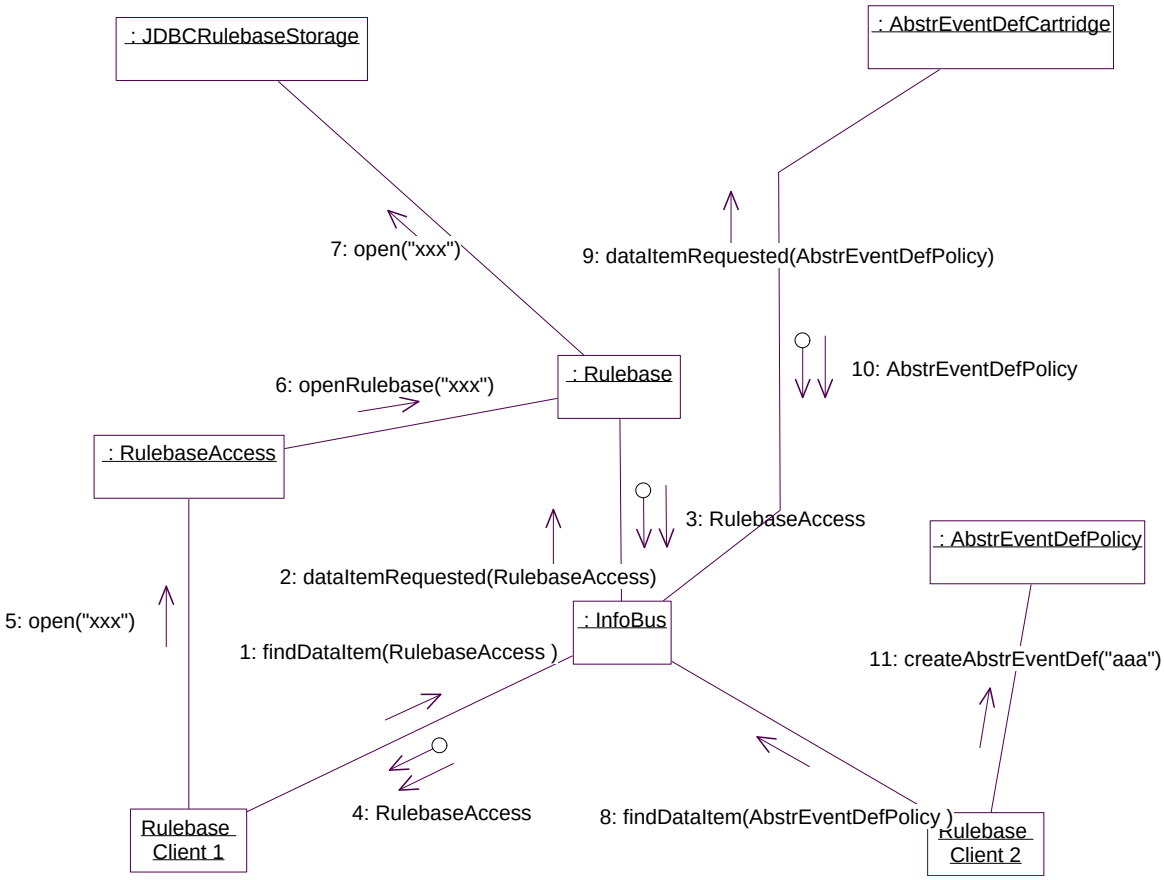

Figure 7.7: Interaction with a Rulebase Component

6. `RulebaseAccess` forwards this request as `openRulebase(''xxx'')` to the rulebase component.

7. The rulebase component establishes a session with the persistence facility by invoking `open''xxx''` of the `JDBCRulebaseStorage` component.

8. Client 2 requests the InfoBus for access to the abstract event definitions (`findDataItem("AbstrEventDefPolicy")`.

9. Since the rulebase is open this request is directly forwarded to the `AbstrEventDefCartridge`.

10. The `AbstrEventDefPolicy` is returned by the Cartridge component.

11. Client 2 initiates the creation of the abstract event ”aaa”.

Note that the rulebase component is not only a data controller but it acts also as a producer by providing the `RulebaseAccess` item.

7.4 The FRAMBOISE Development Framework

The implementation of the FRAMBOISE components, as well as of the component framework is organized as an object-oriented class framework which is called *FRAMBOISE Development Framework* (FDF). The classes of the FDF are divided into the following categories (Fig. 7.8) There are on the one hand three categories that include

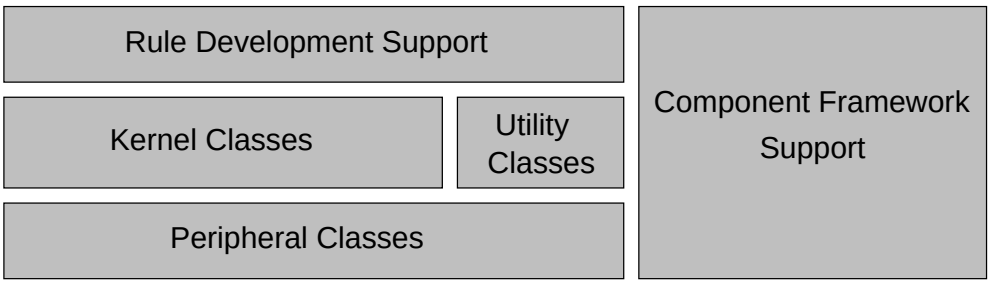

Figure 7.8: Organization of the FRAMBOISE Development Framework

classes used to implement components of an ECA system i.e.,

- The *Kernel Classes* implement the system-independent basic functionality of an ECAS such as rule execution, rulebase management etc. These classes are principally applicable for any ECAS.

- The *Peripheral Classes* provide the functionality to implement the adapters through which an ECAS interacts with a DBMS as well as with the (operating) system environment. Most of these classes are typically specialized for specific DBMSs respectively operating systems.

- The *Rule Development Support* Category includes those classes that are required to provide components that are used to build rule development tools like a rule-browser, rule debugger, rule editor etc.

- The classes to implement the component integration software are summarized as the *Component Framework Support* category.

- Finally there are a number of general purpose classes such as string tokenizers, specific collection classes etc. which are used by all of the above categories. We address them as *Utility Classes*.

Conceiving the FRAMBOISE development framework as a class framework is primarily designed to support rebundling, i.e., to facilitate white box reuse when an ADBI must provide novel components. An implementation overhead in the unbundling process due to the complex class framework design activities has thereby been taken into account in order to foster an efficient ECAS construction.

The design of the FRAMBOISE development framework was investigated in two distinct phases. At the first stage, an initial class structure based on the reference architecture was designed thereby incorporating experiences made during realization of the object-oriented ADBMS SAMOS [GGD+95b].

The initial framework design closely followed the reference architecture described in Chapter 6, i.e., the elements of this architecture were straightforwardly mapped to corresponding classes and interconnection mechanisms (e.g. pipes were implemented as UNIX named pipes). The class framework was implemented in C++ to run on SUN SPARC workstations under SunOS 5.5. (Solaris) thus enabling the reuse of the C++ code written for SAMOS. It consisted of about 150 classes that provide the processing of various primitive event types (abstract and object-related events) as well as the detection and the subsequent processing of complex events as they are proposed in the rule definition language of SAMOS [GD94a]. Furthermore, the framework included a comprehensive rule execution facility, offering various options (e.g. priorities, conflict resolution strategies, cycle policies) to process rules.

According to recommended practices [Sch97], the initial framework was intentionally designed as a rather fixed application[7] and was not yet outlined as a foundation to provide reusable software components in the sense of the FRAMBOISE component model.

Actual component design was performed in a subsequent step. The components were systematically generalized as described above and ported the framework to the

[7] Changeable aspects were identified occasionally and the corresponding hot spot subsystems were designed, however, this was not systematically pursued at this stage.

Java programming language, using the Java Beans component model. Subsequent investigations that based on *vertical prototypes* (i.e., implementing a specific facet completely) relied on this implementation.

Chapter 8

Constructing ECA Systems

The bundling process presented in Section 5.3 defines the principal procedure according to which ECASs are built. It is basically feasible to assemble ECA Systems exclusively with the assistance of general-purpose facilities such as compilers, debuggers, compiler generators (to provide an RDL compiler for the respective ECAS) etc.

Such a method, however, is error-prone, due to the various alternatives of active database behavior. It is easy to conceive that an ADBI overlooks an element (e.g., a specific coupling mode) with the consequence that it is subsequently very difficult to grasp the rule execution behaviour of the operational ECAS. Recalling the inherent difficulties of rule development and maintenance (cf. Sec. 2.4), it is obvious that ECASs were rather a nuisance than value-adding facilities and would even endanger the overall systems where they are deployed.

FRAMBOISE must therefore provide assistance to assert that all aspects of active database mechanisms are taken into account by the ADBI as well as that a rebundled ECAS behaves according to the desired rule execution semantics.

Building component-oriented software systems consists obviously on three basic activities: One must clarify what the overall system shall achieve, one must identify the adequate components and must assembly them together into a coherent system. Thus Section 8.1 presents how an ECAS is specified, followed by the classification schema in Section 8.2 and the procedure to assemble ECA Systems (Section 8.3). Section 8.4 concludes the chapter.

8.1 The Specification of ECA Systems

ECAS construction should not start with the composition of an ECAS right away, since the variety of alternatives of active database behavior is rather complex. Thus, it is appropriate to provide in FRAMBOISE a higher abstraction level that enables an ADBI to “think” for instance in terms of cycle policies for rule execution instead of scheduler components.

The FRAMBOISE *requirements specification language* is a means for an ADBI

to declare specific features of the underlying DBMS and of the system environment and to define the functionality of the prospective ECAS, i.e., the knowledge and rule execution model etc. In this section the chief aspects of the specification language are presented (the complete language can be found in Appendix B). The syntax is expressed in the extend Backus Naur Form (EBNF).

The overall structure of a requirements specification is as follows:

```
ECAS ::=   ‘‘ECAS’’ ECASName DBMSAspects
           KnowledgeModel ExecutionModel
           RuleManagementAspects.
```

8.1.1 Declaration of DBMS Characteristics

Characteristics of the underlying DBMS that must be taken into account for the requirements specification of an ECAS.

```
DBMSAspects ::= "DBMS" Datamodel Datamodel ";"
Datamodel ::=   "DATAMODEL" ("relational" |
                "objectoriented" ) ";"

DML_Statements ::= "DML_STATEMENTS"
                   ("interruptable" |
                   "non-interruptable") ";"
```

Interruptable DML statements are a prerequisite to define a recursive cycle policy (cf. section 8.1.3), because a DML statement must be suspended in order to allow the ECAS to process events which are raised due to the respective DML statement.

8.1.2 The Specification of the Knowledge Model

In order to specify the knowledge model of an ECAS, the ADBI declares which kind of events are to be monitored and under what circumstances conditions are evaluated and actions are executed.

```
KnowledgeModel ::=  "KNOWLEDGE_MODEL" EventTypes
                    ConditionCharacteristics
                    ActionCharacteristics
```

Event types are characterized by their respective *event source* i.e., where the events occur. Events take place in applications or the operating system (*external* events), in the underlying DBMS (*database* events) and finally, in the ECA-System itself (*composite* events). Thus, the event types of the ECAS can be specified by means of the following constructs:

```
EventTypes ::=        "EVENTS" EventSource
                      {EventSource}

EventSource ::=       (External|Database) |
                      "COMPOSITE" ) ";"
```

The event source `Database` is further subdivided. First the *modification* (creation, update or deletion) of a database entity is considered as an event source. According to the data model of the underlying DBMS such entity is either a *tuple* of a relation or an *object* of a class extension. An object-oriented DBMS model may also raise *method events* before or upon a method invocation. Furthermore the transaction manager of the DBMS is also a potential event source. Therefore it is possible to specify that an ECAS processes *transaction events* which are signalled at the start and end of a transaction. Finally, the emission of error messages by the DBMS is another potential (database-) event source. Database error messages have usually a very high priority. Therefore they can also be specially processed by an ECAS. Database events are specified as follows:

```
Database ::= "DATABASE" ("modification"|"method"|
                         "transaction"|"error")
```

Presently FRAMBOISE distinguishes two external event sources: On the one hand the so-called *abstract events* which are explicitly raised by the users. Raising abstract events is also an adequate means to signal application events. On the other hand the system clock is regarded as a particular event source for external events. External event types are accordingly specified:

```
External ::= "EXTERNAL" ( "clock"|"abstract")
```

Condition characteristics indicate when and how conditions are evaluated.

```
ConditionCharacteristics ::= "CONDITIONS" Coupling
                             ConditionEvaluation
```

Conditions can be evaluated immediately upon event detection or at a later stage upon signalisation of a *rule assertion point.* Rule assertion points are either explicitly signalled by a user process or implicitly at the end of a DBMS transaction. Moreover, coupling modes also refer to the relation between the conditions and their eventual triggering transaction. We call the former *temporal coupling* and the latter *transaction binding*.

```
Coupling ::= TemporalCoupling TransactionCoupling

TemporalCoupling ::= "COUPLINGS" "immediate"
                     ["deferred"]";"
```

```
TransactionBinding ::= "BINDINGS" ["current"]
                       ["detached"]";"
```

Finally, the ADBI must specify whether the conditions are queries that are executed under the control of the DBMS and/or if they are functions running outside the DBMS context ('stand-alone').

```
ConditionEvaluation ::= "EVALUATION" ["DBMS"]
                        ["stand-alone"]";"
```

The characteristics of actions are specified analogous to the conditions.

```
ActionCharacteristics ::= "ACTIONS" Coupling
                              ActionExecution
ActionExecution ::= "EXECUTION" ["DBMS"]
                        ["stand-alone"]";"
```

8.1.3 The Specification of the Rule Execution Model

The rule execution model prescribes how an ECAS processes triggered rules.

```
ExecutionModel ::= ''EXECUTION_MODEL'' CyclePolicy
                   [ConflictResolution]
                   RuleConsumptionPolicy
                   TerminationPolicy
```

By means of the parameter `CyclePolicy`, the ADBI defines the rule execution algorithm to be either recursive (i.e., the execution of a rule can be suspended to process events that are signalled inside it) or non-interruptible.

```
CyclePolicy ::= "CYCLE_MODE" ("recursive" |
                              "non-interruptable")";"
```

The conflict resolution defines which rule of a set of simultaneously triggered rules is to be executed next.

```
ConflictResolution ::= "CONFLICT_RESOLUTION"
                       "absolutepriority"
                       "relativepriority"]
                       "FIFO" | "LIFO"";"
```

Event instances can either be consumed locally or globally. Local event consumption implies that event instances execute every rule which is triggered by this particular event. Global event consumption, however, denotes that only one rule is executed, discarding all other rules that are triggered simultaneously. The rule to be executed is selected according to the conflict resolution strategy. Thus, global consumption policy gives the opportunity to overrule the rule execution process, as then the rule with the highest priority is triggered for a single event instance.

```
RuleConsumptionPolicy ::= "RULE_CONSUMPTION"
                          "local"|"global" ";"
```

Finally, the ADBI determines if threads of cascading rules are to be terminated by a limit (which is set and modified at runtime of the ECAS) or if it is granted by the developer of the active application that no infinite threads occur.

```
TerminationPolicy ::= "TERMINATION" ("granted"|
                          "limit")";"
```

8.1.4 Aspects of the Rule Management Model

The ADBI has to provide information whether the rulebase can be modified "on the fly" or whether an ECAS must be halted and restarted in order to activate modifications of the rule schema. The latter might for instance be necessary to enable static linking of external routines that represent an action.

```
RuleManagementAspects ::= "RULE_MANAGEMENT" Adaptability
Adaptability ::= "ADAPTABILITY" ("static" | "dynamic")";"
```

Static adaptability implies that restricted modification of the rulebase is still possible e.g., by recombining previously linked conditions and actions in different rules. However, modified and new rulebase elements are deactivated until the system restarts. In the meantime, the ECAS remains operational, but will emit a warning if such inactive objects are triggered.

8.1.5 Examples

The following example specifies a simple ECAS to process triggers asynchronously as proposed in [SB99a] (cf. Sec. 3.2.5).

```
ECAS ATP
  DBMS
    DATAMODEL relational;
    STATEMENTS non-interruptable;
  KNOWLEDGE_MODEL
    EVENTS
      DATABASE modification;
    CONDITIONS
      COUPLINGS deferred;
      BINDINGS detached;
      EVALUATION DBMS stand-alone;
    ACTIONS
      COUPLINGS deferred;
```

```
      BINDINGS detached;
      EXECUTION DBMS;
  EXECUTION_MODEL
    CYCLE_MODE non-interruptable;
    CONFLICT_RESOLUTION FIFO;
    RULE_CONSUMPTION local;
    TERMINATION limit;
  RULE_MANAGEMENT dynamic;.
```

The following example specifies an ECAS that provides together with an object-oriented DBMS the functionality of the ADBMS SAMOS.

```
ECAS SAMOS
  DBMS
    DATAMODEL objectoriented;
    STATEMENTS interruptable;
  KNOWLEDGE_MODEL
    EVENTS
      EXTERNAL clock;
      EXTERNAL abstract;
      DATABASE modification;
      DATABASE method;
      DATABASE transaction;
      COMPOSITE;
    CONDITIONS
      COUPLINGS immediate deferred;
      BINDINGS current detached;
      EVALUATION DBMS stand-alone;
    ACTIONS
      COUPLINGS immediate deferred;
      BINDINGS current detached;
      EXECUTION DBMS stand-alone;
  EXECUTION_MODEL
    CYCLE_MODE recursive;
    CONFLICT_RESOLUTION absolutepriority;
    RULE_CONSUMPTION local;
    TERMINATION granted;
  RULE_MANAGEMENT static;.
```

8.2 Component Classification in FRAMBOISE

Components must be cataloged in order to be able to identify and retrieve them in an efficient way. By defining the FRAMBOISE component schema (cf. Sec 4.1.5

and Fig. 4.1), it was specified what kind of the information has to been stored and packaged together with a software component. One specific element of this component schema is the so-called *classification information*. This category of information enables component identification and retrieval. We present in this section how components are classified in FRAMBOISE. In the next section, the basic issues of software component classification are discussed. Afterwards, in Section 8.2.2, we present the so-called *faceted classification* that is applied in FRAMBOISE as shown subsequently in Section 8.2.3.

8.2.1 Classifying Software Components

It is actually not yet clear how to specify a software component [Szy97] even though one agrees that catalogue information should principally consist of precise specifications of what components do and what platform requirements they have. Research has concentrated so far on how to catalog and retrieve components but there are no methods that are proven to work with components of substantial complexity. There are various indexing vocabularies to classify components, beginning with generally applicable approaches like free text, keyword classification and enumerated classifications (alike a ISBN number) to more specialized classification schemas that rely on specific facets or attributes (for a comprehensive overview cf. [Sam97]).

Empirical studies [FP94] revealed that there were no significant differences between keyword, attribute value and and faceted classification with respect to search effectiveness and user preferences, even though search times differed significantly.

8.2.2 Faceted Classification

Faceted classification was originally proposed by the Indian mathematician S. R. Ranghanatan [Ran57] whereas a faceted classification schema for the reuse of function-oriented software was described in [PD85, PDF87]. This technique principally intends to classify something along several dimensions that are referred to as *facets*. Each facet can be represented by a set of *terms* with any kind of structure (i.e., words, list of words etc.). In principle a facet is just a multivalued attribute, where there can be some control or structure on the attribute values (terms).

Faceted classification is an adequate method to categorize software components, because it allows to classify the components according to several aspects. It, however, is not trivial to determine which characteristics should be represented by facets. To choose suitable facets for classifying a set of components, the following steps must be performed [Kar95]:

- Construct an overview of the components of interest.
- Determine which aspects of a components are most important for retrieval.

Thereby one should take the following principles into account:

- *Expressiveness*. Each facet must say something about the component that is interesting to the person wanting to retrieve it.
- *Tractability*. There must be no more facets than a developer can cope simultaneously with it. Three to six facets are considered as appropriate.
- *Conciseness*. Each facet should be well defined. There should be no ambiguity about which facet covers which aspect of a component and the facets should be orthogonal to each other as possible.
- *Relevance*. All facets should be relevant for most components, regardless of their size, form, internal structure or contents.
- *Ease of Construction*. It should be as easy as possible to construct a term space for the chosen facets.

Faceted classification has been applied to classify specific DBMS kernel services [Gep94] as well as to classify event based systems [Tom99].

8.2.3 Application for FRAMBOISE

We classify components in FRAMBOISE with the following five facets:

Abstraction: This facet describes what a component is. Usually a component can be described by a noun that stems from the vocabulary of the reference architecture, e.g., an event signal processor.

Target DBMS: Indicates for which specific DBMS a component is applicable.

ADBS Functionality: Indicates the knowledge and the execution model for which the component is applicable. It bases on the vocabulary of the specification language.

Rule Management: Describes how rules are managed. It bases likewise on the vocabulary of the specification language.

Constraints: This facet lists dependencies (platform and other) which constrain potential reuse of the component, e.g., Unix-based, requires JDBC etc.

A facet is a set of attributes that apply for a specific component as shown in the examples in table 8.1. It is feasible that a facet of a specific component contains no terms. Such an empty facet is considered as `null` value, i.e. there is no information available. The letter * indicates that every attribute applies for this facet. For instance the working memory in table 8.1 is applicable for every DBMS whereas it is not known whether there are restrictions concerning the rule management. Note that the DBMS *Cloudscape* [Clo], that is addressed in table 8.1 is an object-relational

Abstraction	Target DBMS	ADBS Functionality	Rule Management	Constraints
Working Memory	*	CYCLE_MODE recursive TERMINATION limited BINDINGS deferred		Green Threads
Event Storage	Cloudscape	EVENTS database	ADAPTABILITY runtime	JDBC
Event Detector	Cloudscape	EVENTS complex	ADAPTABILITY runtime	C++ VERSANT SOLARIS

Table 8.1: Example Classification of Components

database management system that is written entirely in Java. It has originally been devised by Informix.

Experiences with faceted classification method indicate that one should not rely exclusively on this technique to retrieve components. For instance [PY93] advocates a combination of classification methods integrated with text searching techniques and hierarchical ordering of the facets. Thus the component retrieval procedure with methods for glass-box reuse that enable the identification of reusable software through source code analysis are supplemented. Even though this so-called *code-scavenging* is rather an ad-hoc and unsystematic procedure, it is considered as a quite effective approach to reuse software [Kru92]. Recalling that the commercial feasibility of the open-source model [Ray01] has been proved so far, glass box reuse is in FRAMBOISE also viable from an economic perspective.

8.3 The Assembly of ECA Systems

An effective construction activity depends also from the ease of the system assembly and an adequate tool support. The momentum in the Java industry yielded various sophisticated software development environments of excellent quality that enable among other things the visual assembly of applications out of preexisting Java Beans. It is therefore pointless to investigate thoroughly into the provision of development environments for FRAMBOISE. Instead we discuss in this section what kind of tools are necessary to support an ADBI in the assembly process that goes beyond the scope of these general purpose tools.

ECA-Systems are basically assembled according to the following procedure (see Figure 8.1): First, the ADBI specifies the system requirements of the ECAS to be

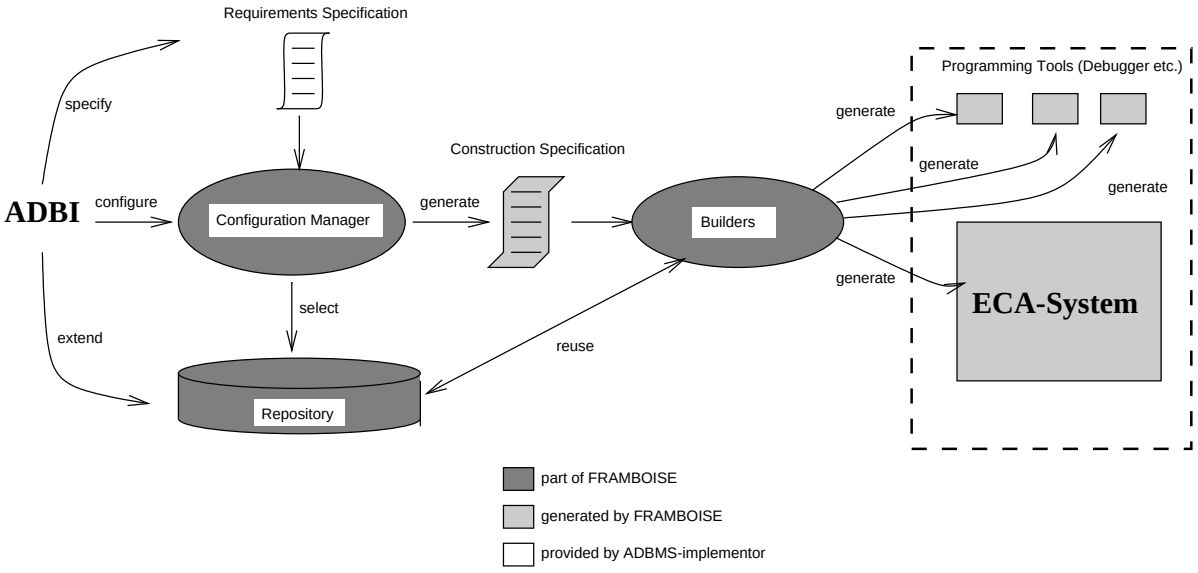

Figure 8.1: Construction of an ECA-System with FRAMBOISE

built by means of the *requirements specification language* of introduced in section 8.1. The ADBI is supported here by the so-called *configuration manager*. This tool first processes the requirements specification in order to query the repository for candidate artifacts and presents the result to the ADBI. The configuration manager, offering features to browse the repository, allows the ADBI to determine which artifacts are directly applicable and are accordingly incorporated into the ECAS and which artifacts are to be specialized or must even be realized from scratch. Afterwards, the ADBI provides the missing artifacts and makes them available for the further construction process by integrating them in the repository. Prototypical implementations of these tools are presented in Appendix C

When the ADBI has finished the configuration process, the configuration manager generates the *construction specification* of the ECAS. This document is used as input to various builder tools which generate the operational ECAS, a rule definition language valid for the respective ECAS and accordingly a rule definition language (RDL) compiler with a rule browser and a rule editor specialized for this ECAS. Hence, the construction specification is not a functional description like the requirements specification, but consists of lists of identifiers indicating concisely which components form the final ECAS.

The form of the construction specification depends on the technology chosen to realize the components. With this separation into requirements and construction specification that resembles the classical division of a compiler into the front- and the backend (cf. [ASU86]) it is relatively easy to provide the FRAMBOISE specific tools for different software development environments. It is for instance feasible to generate grammars that are processed by the Unix ”make utility” in order to build an operational

ECA Systems. Correspondingly, it is feasible to generate files that are applicable for specific Java Development Environments.

8.4 Conclusion

This chapter presented which ADBMS-specific facilities are required to perform an effective rebundling of ECA Systems. We established a specification language that serves also as a foundation to classify and retrieve components. The vocabulary introduced thereby enables an ADBI to query the component repository according to the ADBS specifics of the prospective ECAS. Finally the kind of tools that are necessary to support an expedient assembly of the respective ECA System were sketched.

Chapter 9

Evaluating FRAMBOISE

In this chapter, it is evaluated whether our conception of a construction system enables the cost-effective provision of active database functionality for arbitrary DBMSs. First, the adequacy of the ECA Systems is assessed in Section 9.1 by evaluating its reference architecture. In order to evaluate the versatility of FRAMBOISE, Section 9.2 evaluates how the construction system combines with techniques to furnish passive database mechanisms. Subsequently, in Section 9.3, the evidence of a cost-effective construction activity is discussed. Finally, Section 9.4 concludes the evaluation.

9.1 Evaluation of the Reference Architecture

It is nowadays agreed that architecture design is too complex that one could establish a simple set of quality attributes that can be applied mechanically to assess the quality of an architecture. The main challenge of architecture design is the striving for balances between understandability of the design, achievement of the required functionality and business aspects (e.g., time to market, projected lifetime of a system etc.) [BCK98]. These typically conflicting interests must be weighted differently from case to case in order to assess an architecture. Hence we will discuss in Section 9.1.1 the coherence of the reference architecture, in Section 9.1.2 how it supports the bundling-oriented construction procedure devised for ECASs and finally, in Section 9.1.3, the consequences of the reference architecture for operational ECA Systems.

9.1.1 Coherence

An architecture obviously must be complete and correct to meet all requirements and runtime constraints of the respective system. Both criteria are fulfilled in our context. Completeness has been achieved by transforming a comprehensive model of unbundled ADBMSs systematically into the reference architecture of ECASs. The adoption of a formal architecture definition language to describe the architecture enabled us to

reason about the anticipated behaviour of the system and the correctness of the architecture.

The assurance of completeness and correctness is mandatory but not sufficient to achieve proper quality. In that sense, the achievement of *conceptual integrity* is considered as the most important architecture quality attribute [Bro95]. This means that there must be an underlying theme or vision that unifies the design of the system at all levels – the architecture should do similar things in a similar way in order to be easy to grasp. Understandability, however, is very difficult to measure; it has been advocated that each level of an architectural description should consist of three to seven entities, because the human perceptive system is naturally able to comprehend aggregates of up to around seven entities simultaneously [BAMS93, Bro84]. We feel that we achieved a satisfying conceptual integrity for the reference architecture by applying the architecture styles as patterns to simplify and streamline the design. Moreover, the organization of an ECAS as an interpreter indicates good understandability, because it provides an architectural level that enables an ADBI to cope merely with a restricted set of components to solve a specific problem.

9.1.2 Architectural Qualities Discernible at Construction Time

The effectiveness of the FRAMBOISE approach depends to a large extent on the ability to make separately developed components of the system work together correctly (so-called integrability [BCK98]) and on the ability to reuse the structure of ECA Systems or some of their components in future ECASs. Furthermore, the reference architecture forms a basis to get subsystem and component design off ground.

System Integrability

The integrability of a system depends on the external complexity of the components as well as their basic interaction mechanisms and protocols. Since a software architecture determines the partitioning of a system, it influences these aspects directly. The reference architecture contributes to the integrability of the ECA Systems as the chosen architecture styles enable us to design highly cohesive components that collaborate with a few other components by means of precise interaction patterns. Distinct tasks of active database functionality such as rule execution cycle, condition evaluation etc. are placed into distinct components so that a component has typically just one specific responsibility. Thus the reference architecture prepares the ground to specify narrow and concise component interfaces which promote integrability further.

Reusability

The reference architecture is designed up front for reuse by enabling the construction of a broad range of ECASs. Therefore the reusability of the reference architecture is

not a primary issue to be qualified[1], but the reusability of the architectural components. On an architectural level, the reusability of a component depends on how tightly coupled it is with other components. The components in the reference architecture are rather loosely coupled even though most of them interact with the rulebase. However, due to the chosen blackboard architecture style, the rulebase invokes its associated components implicitly. Besides being related to the rulebase, each component must know about maximally one other component. A component in the pipeline does not even know the identity of their upstream and downstream filters.

Subsystem Design

The reference architecture is also a means to determine concrete software components and their implementations. Even though the component provision is dominated by technical features, it is foreseeable that architectural aspects will remain an issue. Components and connectors are decomposed into further subcomponents and the design of new elements (e.g., system-specific connectors) must be checked for properties such as the local absence of deadlocks. The strong cohesion of the components and their loose coupling induced by the reference architecture facilitates the design of subsystems because the designer can mostly focus on local architectural decisions. The architecture description language WRIGHT is for its part a useful vehicle to support the further 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. Correspondingly, the reference architecture can be transformed into system architecture specifications of specific ECA Systems.

9.1.3 Architectural Impacts on Operational ECA Systems

ECA System should interfere as little as possible with the encompassing applications and the DBMSs. Therefore, once an ECAS is in operation, it must process event signals and execute rules in a most efficient manner in order to minimize blocking of the triggering transactions. Furthermore the ECAS must be highly available in order to prevent that the system formed by the applications, the DBMS and the ECAS crashes on account of the latter's failure. Finally, operational ECA Systems may undergo revisions due to maintenance activities so that its modifiability must be considered also at the architectural level.

Performance

High reusability and integrability do not come for free. The isolation of functionality that makes pipes and filters so modifiable often leads to poor performance results. Filters typically force the lowest common denominator of data representation (e.g., an ASCII stream). Should the input stream be transformed into tokens, every filter

[1] In fact, only the modifiability of the architecture is of interest; cf. Sec. 9.1.3.

pays this parsing/unparsing overhead . If a filter cannot produce its output until it has received all of its input, it will require an input buffer of unlimited size, because bounded buffers could cause deadlocks. Finally, each filter operates as a separate process or procedure call, thus incurring some overhead each time it is invoked. We chose this architecture style despite of these shortcomings, because they can be coped with in the following ways: By means of typed pipes we can mitigate the parsing overhead to some extent. Furthermore, we can assert that all associated tokens (i.e., event and rule instances) come along without delay so that a filter should not wait overly long in order to process them together. Finally, it is conceivable that an event or rule instance does not contain large data structures, thus enabling rather efficient forwarding.

Availability

Typical architectural techniques to ensure high availability of a system are the installation of redundant componentry that takes over in the case of failure, careful attention to error reporting and the provision of special components such as time-out monitors. For the following reasons none of these issues is adequate for the reference architecture of ECAS. Providing a priori redundant components *within* an ECAS might compromise efficient rule processing, whereas the redundant installation of complete ECAS is rather an aspect of the system architecture of a concrete system. Error reporting depends on logging or monitoring facilities whose incorporation into a system is easy to achieve (e.g., as additional filters in the pipeline), but for the sake of clarity they are omitted here. Nevertheless, the careful separation of concerns applied in the reference architecture promotes a good testability and reusability which enable the provision of a less error-prone system that in turn lengthens the mean time to failure.

Modifiability

From an architectural viewpoint it is relevant whether maintenance activities will precipitate a modification of one component only, of more than one component or of something more drastic such as a change of the underlying architectural style. Due to the loose coupling of the components and the clean partitioning of the functionality, chances are high that modifications can be traced down to single components as long as the rulebase is not affected. Adapting the rulebase will affect practically all components of an ECAS which may often result in a complete reconstruction of the ECAS. We consider this specific case as acceptable, because such a modification is usually induced by a revision of the knowledge model of the ECAS which in turn suggests that the most fundamental requirements of the respective ECAS change.

However, the modifiability of an ECAS has one severe restriction insofar as it is not feasible to introduce components as filters in the pipeline that depend on the state of another filter, because there is no way for filters to cooperatively interact to solve a problem. In such a situation one is required to replace the pipes and filter style by

another architecture style. We consider the risk that such a drastic change occurs as small, because the pipes and filters architecture corresponds very well to the batch mentality devised for rule processing [PD98].

Nevertheless, the question remains to what extent the reference architecture itself is modifiable in order to cope with future requirements. Actually, such a revision implies not only the introduction of one or several new components, but the application of different architecture styles. In such a case, the designer can fall back to the conception of an ECAS as an interpreter and perform an architecture refinement like the one exercised in this chapter. Hence the virtual machine architecture provides a meta architecture that assists an ADBI to adapt the reference architecture of ECA Systems to specific purposes.

9.2 Constructing Active Database Management Systems with FRAMBOISE

ECA Systems are not ADBMSs but provide active database functionality in conjunction with mostly passive DBMSs. In order to evaluate whether it is feasible to furnish ECA Systems for a broad range of DBMSs, we discuss in this section how FRAMBOISE combines with the various approaches of Component DBMSs (CDBMS), relying on the classification of component DBMSs elaborated in [DG00]. Since it is feasible to enhance nowadays commercially available DBMSs to some extent, they are also covered by this classification.

9.2.1 Plug In Components

This category of CDBMSs comprises so-called *universal servers*. The core functionality of such a system is principally formed by a fully functional DBMS that implements all standard functionality expected from a DBMS. Nonstandard features or functionality not yet supported can then be plugged into this “core DBMS”. The DBMS architecture defines a number of plugs through which the services of the respective components are invoked and that the component must implement.

Implementing with FRAMBOISE an ECAS that acts as a plug-in component providing active database functionality is principally easy to conceive. The various adapters of an ECAS (e.g. event detection, action execution etc.) implement then the respective plugs. However, in practice the opportunities to provide ECAS are limited. To date all systems in this category are based on the relational data model and existing relational DBMSs such as IBM’s DB2 UDB [IBM95], Informix Universal Server [Inf98], Oracle8 [Ora99] and Predator [Ses98]. The components in this kind of CDBMS are typically families of base and abstract data types or implementations of some ”traditional” DBMS function such as new index structures. To our very best knowledge none of these DBMSs provides any specific plugs to provide active database functionality

that goes beyond the trigger facilities that are nowadays actually an SQL standard.

As a consequence, it is difficult to implement ECAS that provide rule execution within the triggering transaction (e.g.,immediate or deferred coupling modes). In principle one has to implement workarounds that block the triggering transaction and communicate when to unblock and which operation must be executed as next. Even though such "tricks" are feasible, they usually bear the risk of dangerous side effects such as blocking due to a deadlock or even crashing the DB server. In fact, it was possible to implement immediate and deferred rule execution for a well-known relational DBMS by exploiting its trigger mechanisms, event signalling and the interprocess communication facilities. However, we experienced also the downside, i.e., side effects that blocked the database server or unexpectedly aborted the triggering transactions, so that we wouldn't easily recommend this technique in a productive environment.

Nevertheless, FRAMBOISE can add value to nowadays plug-in CBDMSs. As shown in [HK97, SB99a] it makes sense to provide rule execution engines that process ECA rules asynchronously. Since virtually all DBMSs offer asynchronous event notification, it is feasible to provide an ECAS with a functionality corresponding to the one of TriggerMan [HK97], whereby such an ECAS can provide additionally complex event detection.

However, the provision of plugs to add active database mechanisms that enable synchronous rule processing is not an unrealistic idea. Nowadays object relational DBMSs furnish a considerable amount of functionality to provide such plugs. For instance extension mechanisms to integrate third-party program routines into the DBMS such as the function manager of Oracle 8 [Ora99]. These facilities provide a safe execution environment that allocates specific (eventually DBMS external) address space to these routines and restrict their access to specific resources. Such an environment to execute external programs in the context of a database transaction could also serve as a basis for plugs to add active database functionality.

These plugs correspond basically to the database event detection, condition evaluation and action execution connector. Thereby action execution and query evaluation takes place under control of the core DBMS, eventually by involving user-defined functions in a safe execution environment as described above. This "sandbox" must additionally be able to handle time-outs in order to unblock the triggering transaction if the rule execution engine does not respond timely.

The format of the information (e.g., transaction context or user authentication) to be interchanged between the core DBMS and the rule execution engine can rely on standards. For instance, the *JavaTM 2 Platform, Enterprise Edition* standard [Java] for developing multi-tier enterprise applications (J2EE) specifies standard interfaces between a transaction manager and the parties involved in a distributed transaction system. Similar specifications are used to encapsulate authenticated user information. Hence the provision of "active database plugs" for an ECAS-like facility can to a large extent rely on techniques that are already available.

9.2.2 Database Middleware

These approaches are devised to integrate existing data stores into DBMSs. Thereby the data items are left under control of their original management systems while they are integrated into a common-style DBMS framework. It is, for instance, feasible to integrate existing data stores into query processing or transaction management of an entire system.

Since the systems integrated by means of the database middleware will typically exhibit different capabilities, (e.g., query languages with varying power) the basic problem of database middleware is that it needs to understand the data formats and functions of each data source. Thus the basic architecture of database middleware introduces a common intermediate format into which local data formats can be translated. Components are introduced that are able to perform this kind of translation. Furthermore, common interfaces and protocols define how the database middleware and the components interact (e.g., in order to retrieve the data from a data store). These components (usually called wrappers) are also able to transform requests issued via these interfaces (e.g., queries) into requests understandable by the external system.

FRAMBOISE can be applied to furnish active database functionality as database middleware, because ECA Systems are principally outlined to provide active database functionality for different data models and different data sources. The system-specific idiosyncrasies are transformed by means of the respective connectors (e.g., DB event detection) into the ECAS internal representations. Thus these connectors act effectively as wrappers.

It is furthermore conceivable to provide ECAS that rely directly on the common format provided by the standards of the database middleware. For instance OLE DB [Bla96] provides an open and extensible collection of interfaces that enable applications to have uniform access to data stored in DBMS and non-DBMS information containers. OLE DB, based on the Microsoft Component Object Model (COM), defines the system-level programming interfaces that encapsulate various DBMS components. These interfaces extend Microsoft's OLE/COM object services framework with database functionality. The OLE DB functional areas include data access and updates (so-called *rowsets*), query processing, schema information, transactions, security and *data source notifications* (watches).

The latter are designed to enable clients to be notified about changes to the underlying data source, originated by other concurrent clients running under different transaction contexts in either read committed or read repeatable isolation levels[2]. Upon a notification, the client can request a list of changes from the respective data provider. Applications of data source notifications are, for instance, the support of replicated data or materialized views. The client code may then represent an incremental refresh algorithm that is triggered by the notification.

[2]Data source notifications under read uncommitted or serializable isolation levels are not feasible in order to prevent unpredictable results or to avoid that uncommitted data is read by concurrent transactions.

Notifications in OLE DB define a basic mechanisms on which to implement active database behaviour. Due to the strong transaction isolation levels of the data source modifications it is, however, only feasible to implement rule execution that corresponds to the causally dependent coupling mode. This is, after all, not a major restriction in this context, recalling that the main purpose of database middleware is the integration of various, principally independent data sources. The data sources themselves should not be modified. Hence database middleware does not introduce active database functionality in the data source. Active database behavior should be executed in the integration layer formed by that the middleware. Thus the causally dependent coupling is practically the only coupling mode that makes sense.

Providing an ECAS that interoperates with an OLE DB environment is quite a straight-forward procedure. The abstractions provided by the OLE DB enable an ADBI to infer the knowledge and rule execution model of the prospective ECAS. For the actual implementation of the ECAS one can rely on commercially available software building blocks, that bridge between OLE DB and the Java virtual machine. Hence it is not overly difficult to map the idiosyncrasies of OLE DB within the respective connectors to the FRAMBOISE specific counterparts.

9.2.3 Database Services

This type of componentized DBMS is characterized by a service-oriented view of database functionality [GD98]. All DBMS functionality is provided in *standardized* unbundled form. Plugs are service definitions, components are service implementations.

A prominent example are the CORBA Services [Gro95] which leverage several DBMS tasks to general object systems. The services are standardized by the Object Management Group (OMG) which was formed in 1989 as an industry consortium, with the aim of addressing the problems of developing interoperable, reusable and portable distributed applications for heterogeneous systems, based on standard object-oriented interfaces. These problems are addressed by introducing an architectural framework with supporting detailed interface specifications. OMG's role is that of an interface and functionality specifier; it does not develop software itself.

The reference model identifies and categorizes the components, interfaces and protocols that constitutes the Object Management Architecture (OMA). As depicted in Figure 9.1 there are four categories of components, namely *object services*, *common facilities*, *domain interfaces* and *application interfaces*. These are linked by an object request broker (ORB) component which enables transparent communication between clients and objects. Domain and application interfaces are not relevant in the current context, because they provide abstractions for various domain-specific application domains or even external application interfaces that users develop. Such domain specific standards are also referred to as *vertical standards* [DW99], contrasting to the so called *horizontal standards* that establish a domain independent, common mechanism for basic services such as security, transactions etc. The horizontal standards are defined in

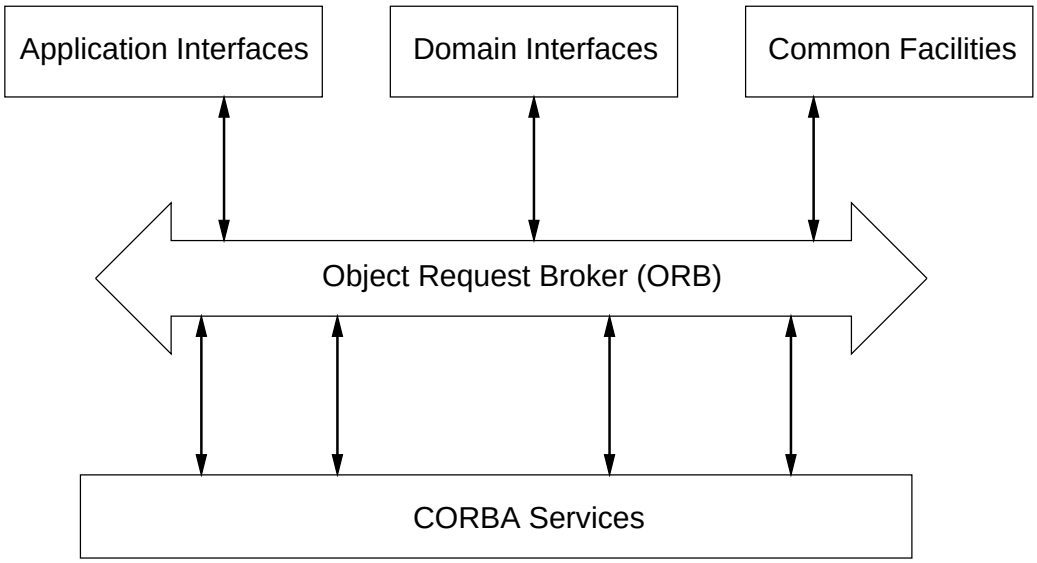

Figure 9.1: Object Management Architecture

the CORBA Services and to some extent in the common facilities. The former principally provide value-added services to facilitate the development of distributed applications. Among others, CORBA services comprehend classical database services such as persistence, transactions, concurrency control, queries and even (asynchronous) event notifications. In theory it is feasible to put together a component DBMS by gluing together various CORBA services.

It suggests itself to situate here also a service for active database mechanisms and – in our context – to implement it by means of FRAMBOISE as an ECAS. Indeed the OMA proposes also a *Rule Management Facility* which is, however, not considered as an CORBA (object) service but is situated in the *common facilities*. This group of services is formed by four major complexes: *user interface*, *information management*, *system management* and *task management* [OHE96]. The latter is in turn decomposed into several facilities, one of them is the rule management facility. It is defined as follows "The rule management facility provides for declarative event-condition-action rule specification and processing. Rule management involves the acquisition, management and execution of a rule" [tas95]. The proposal refers explicitly to ADAMS functionality but includes also features like inference mechanisms, deductive rules and certainty factors.

Even though FRAMBOISE is not conceived to provide inference mechanisms, it enables the provision of an ECAS that implements quite a powerful rule management facility. In principle, such a "CORBA ECAS" has to provide event detection connectors that interoperate with the CORBA event service, condition evaluation connectors that are specific for the CORBA query service, whereas the action execution connectors invoke operations on CORBA objects. The CORBA transaction and concurrency

services enable the ECAS to perform rule execution within a transaction context. Finally, the rulebase component of the ECAS can rely on the CORBA persistence service to manage rulebase persistence. An ECAS does not represent a mere rule execution engine, but comes along with various tools to define rules and manage rulebases as discussed in Chapter 8. Finally, the component model chosen for our implementation is also applicable in a CORBA environment since Java/CORBA Bridges are also available.

As far as it is known, the proposal of the rule management facility never went beyond the inception phase, i.e., there has never been anything more than a rough outline of this facility. In that situation, the FRAMBOISE construction process enables an ADBI that intends to provide an ECAS that implements a rule management facility, to fill the gap between ”wish list” [tas95] and the ramifications of a concrete implementation (e.g., rule execution models which are not at all addressed in this document, coupling modes and their significance for the interaction with the CORBA transaction service etc). FRAMBOISE gives the ADBI a means to analyze in detail what kind of rule functionality is required and what the implications fur the prospective system are. Hence it is feasible to achieve a detailed specification of an ECAS in a way as it is not provided by the OMG proposal.

9.2.4 Configurable DBMSs

Configurable DBMSs are similar to the DBMS services addressed in section 9.2.3. They rely accordingly on unbundled DBMS tasks that can be mixed and matched to provide specific database support. However, configurable DBMS take this idea one step further because the set of services offered by them is not standardized and fixed. Instead it is planned from the outset to adapt service implementations to new requirements or to define new services whenever needed. As a consequence, components that provide the same DBMS task can vary not only in their implementations for the same interface, but also in their interfaces for the same tasks.

The application of FRAMBOISE in such an environment is straightforward. On the one hand the reference architecture (cf. Sec. 9.1.3) ensures that ECA Systems can be adapted upon modifications in a configurable DBMS.

On the other hand, the overall construction system FRAMBOISE is compatible with the principal techniques of configurable CDBMSs. The construction process that defines how to proceed in order to obtain a (configurable) DBMS with the desired functionality typically consists of several phases including requirements analysis design, implementation and integration of multiple subsystems. For each type of subsystem a dedicated construction process is defined and integrated into the enclosing DBMS-construction process. The instance bundling process as described in Section 5.3 corresponds effectively to the construction process of such a subsystem and can be integrated in the enclosing DBMS-construction process. Thereby rule extraction and constraint analysis of the FRAMBOISE process will be part of the overall requirements analysis phase, whereas system integration will become part of the embracing

system integration.

Typically for the construction of each subsystem, a dedicated specification language is used to define its functionality. These specifications serve as input to subsystem-specific implementation phases, which in turn use techniques such as the generation of subsystems or the configuration of subsystems out of reusable already existing components. The FRAMBOISE specification language (cf. Sec. 8.1) fully corresponds to such a subsystem-specific specification language.

Finally the complete set of methods to engineer components are also part of the bundling process (cf. Sec. 5.3). Thus FRAMBOISE enables an ADBI to (re-) design an ECAS in depth in order to cope with novel requirements at a configurable DBMS. Thus it is a realistic vision that FRAMBOISE might become part of a DBMS construction kit that enables the provision of configurable DBMSS.

9.3 Cost-Effectiveness

Evaluating the cost-effectiveness of a construction system that exists exclusively as a research environment is a rather speculative endeavor. In fact, our research prototypes were neither developed nor used within budged constraints that would surface cost-*in*effectiveness mercilessly. Nevertheless it is adequate to discuss some key issues that give evidence whether our approach principally enables a cost-effective construction of active database functionality.

Since the provision of CDBMSs is never a simple endeavor, it is a fair assumption to require skilled software engineers for that task. Under that circumstances, the cost-effectiveness of a DBMS construction system depends on the effort an engineer has to make in order to master such a highly specialized technology.

The unbundling process enabled us to furnish a prototypical implementation of the component framework (cf. Appendix C) with quite limited human resources within a controllable schedule. The fact that the lion's share of the prototype has been implemented by one single person made it virtually impossible to exercise extreme programming (XP) as proposed in Section 5.2.2. Aspects of extreme programming that are exclusively related to human interaction (e.g., pair programming, collective ownership of code) are of paramount importance in the overall XP methodology, but they were simply not applicable in our context. It was, however, feasible to follow the XP development cycle (cf. Sec. 5.2.1) according to the modifications proposed in Section 5.2.2. For instance the so-called *planning game* [3] underlay the prototype implementation activities, whereby the reference architecture (Chap. 6) were effectively useful as a system metaphor. The programming tasks were implemented by consequently implementing the unit tests up front by means of the *JUnit test framework* [obj] thus enabling a constant code-refactoring. As a consequence, it could be verified that it is effectively practical to map the construction process of an active database construction

[3] an XP strategy to maximize the value of software produced by a team under uncertainty [Bec00]

framework onto the XP methodology. Thus it can be inferred that it is actually feasible to furnish an active database construction system according to the procedures proposed in this thesis in a cost-effective way.

The following key elements of FRAMBOISE are a basis that ensure subsequently a cost-effective construction of ADBMSs: By means of the reference architecture, the specification language and the component specification that relies on the component schema, provide a set of interlocking concepts that enable a software engineer to quickly grasp the essence of active database technology as it is furnished by FRAMBOISE. Furthermore, the bundling process and its associated techniques give the ADBI a guideline to build an ECAS and how to cope with unforeseen issues (by referring to the unbundling process). Such a road map is a further prerequisite of a cost-effective software provision.

Finally, software reuse is a central theme of any cost-effective construction system (cf. Sec.3.3). Software reuse occurs in FRAMBOISE at all levels of abstraction:

1. At the *component framework level* by reusing artifacts that apply for entire ECAS such as architectures, specifications, test cases, generators that bundle ECAS etc.

2. At the *component level* by deploying prefabricated components.

3. At the *implementation level* by reusing software building blocks like modules, class or function libraries, source code etc. which are reused to implement components.

Our experiences underpin the assertion of that ECA Systems can be constructed cost-effectively. In order to check the development cycles of the component framework, we used FRAMBOISE to realize operational ECA-Systems for three commercial DBMSs. The requirements specification for each ECAS has been established by characteristics of the appropriate DBMS and by the required functionality for the knowledge and execution model. This functionality is drawn from a financial application[4] requiring advanced active functionality. The experiences were very encouraging, as the effort to built the ECASs offering advanced functionality similar to SAMOS was small compared to building SAMOS initially.

Implementing these applications, we experienced that the framework and the operational ECASs proved to be an effective complement of the proprietary event notification/detection mechanisms provided by the commercial DBMSs. Applying the respective raw event notification facilities burdens various tasks to the application developers, e.g., programming demon processes to perform blocking reads of the event queues, event signals are to be numbered, user-defined information is to be packed in strings that are part of the notification etc. Much of this programming effort could

[4]This application had initially been developed for our ”home-made” active object-oriented DBMS SAMOS [GGD95a] aiming at the demonstration of the rich event handling and rule processing facilities of SAMOS.

be significantly reduced because the respective operations were embedded in specialized components. The subsequent reuse of these system-specific enhancements was remarkable. Moreover, FRAMBOISE added value to the respective event notification facilities as they are seamlessly integrated in a powerful rule processing facility i.e, the respective ECAS, that gives the opportunity to combine (and recombine) the event notifications in a uniform way with the execution of various activities. Thus, we experienced that the overall procedure to specify and construct an ECAS leveraged a considerable amount of know-how in active database technology to these mostly passive DBMSs.

9.4 Conclusion

The feasibility of the approach elaborated in this thesis has principally been verified by means of prototypical implementations (cf. App. C). They rely on the JavaBeans™ [Javb] technology so that component prototypes were implemented as Java Beans and were in turn connect by means of InfoBus as discussed in Section 7.3.4. Further facilities of the JavaBeans component infrastructure enable component developers to furnish specific (visual) builder tools to inspect, customize and connect the Java Bean components they develop. Accordingly a component repository (cf. Sec. 8.2.3) prototype was implemented as well as tools to assemble components into coherent ECASs (cf. Sec. 8.3).

Henceforth (equally prototypical) ECA Systems were provided, furnishing active database functionality as plug-ins (cf. Sec. 9.2.1) for nowadays commercially available, mostly passive DBMSs. By means of this prototypes it was feasible to proof that it is basically feasible to furnish active database functionality by means of a component framework as proposed in this thesis. The prototypes gave furthermore evidence for the cost-efficiency of this approach as discussed in 9.3.

However, prototypes have their limitations. For instance, due to the complexity of even a prototypical implementation of the component framework, it is virtually impossible to assess whether eventual runtime inefficiencies are to be attributed to the component infrastructure (e.g., Java virtual machine), simple deficiencies in the prototype code or whether they are in fact an architectural weakness. The evaluations made in this chapter are therefore complementary to the prototypes to fully demonstrate the viability of FRAMBOISE.

Hence we conclude that our goals have been achieved, because it is feasible to implement a component framework according to the principles elaborated in our work. The reference architecture of the ECAS leverages the implemented software prototypes into effective ECA Systems by fostering the sound, flexible and comprehensive provision of active database mechanism (Sec. 9.1). FRAMBOISE is equally versatile enough to be applied in a broad range of scenarios as discussed in Section 9.2. Finally there is evidence that the construction of the ECA systems mechanisms can be achieved in quite a cost-effective manner.

Chapter 10

Conclusion

This chapter concludes the thesis. It gives a summary, identifies contributions of the thesis and describes open issues and directions for future work.

10.1 Summary and Contributions

This thesis investigates in the systematic provision of sophisticated active mechanisms (so-called ECA Systems) in database or database-related environments and proposed an engineering approach – named FRAMBOISE [FGD98] – to construct them in a cost-effective way. Regarding that there is no actual construction theory in the domain of active database technology, the detailed elaboration of such a construction system is new to this field.

Unlike other approaches [BDD+95, HK97, BFL+97], FRAMBOISE conceives the provision of active database facilities as a software engineering process and addresses all relevant phases for the construction. Approaches that start with a toolbox approach right away or offer a fixed architecture are less likely to succeed in terms of high reuse pay-off. The thesis identified and specified three major processes and modeled them in detail, namely

1. a process that drives the provision of the construction system itself (i.e., meta bundling process),

2. another one (instance unbundling process) to consolidate the variety of active database technology into a coherent ensemble and to furnish the building blocks of the subsequent systems.

3. Finally a guide to the actual construction of the active database services has been established by means of the instance bundling process.

The activities performed according to the meta bundling process, enable the identification of a component-based approach as the appropriate technology and to establish a concise notion thereof. Regarding the ambiguous conception of component software

that exist nowadays, clarifying the application of this paradigm for a specialized context is a further contribution. The same applies to definition of a specific architecture model that has equally been performed as a a specific activity of the meta bundling process.

A major contribution of this thesis is the systematic unbundling of ADBMSs in order to gain reusable software components. Thereby the following achievements were made:

- A *reference model* of unbundled ADBMSs was installed and transposed systematically into a *reference architecture* of an active database service.
- Due to the formal specification, the reference architecture was verified for consistency and served as a detailed foundation for the subsequent component provision.
- A procedure to systematically generalize software components was devised.
- A substantial amount of the components was implemented prototypically, using the Java Beans™ [Javb].

In order to ensure an efficient construction of active database services, the thesis proposes

- a language to specify the requirements of the ECA Systems,
- a schema to classify the software components in order to retrieve them according to ADBMS specific requirements and
- an environment to support an ADBI in the assembly of an ECA System.

The approach elaborated in this thesis enables the construction of of active database services that interoperate with a wide range of systems, namely

- with *commercially available*, mostly passive, database management systems, as well as
- with novel approaches to implement *DBMSs which have equally a componentized architecture* and allow users to add components.

Finally, the construction system FRAMBOISE represents a *fully-fledged component framework* situated outside the domain of graphical user interface building. Since such component frameworks are *still rare* [Szy97, Mau00], the systematic procedure applied in this thesis to work out a precise and detailed set of interlocking concepts to specify, design, verify and classify software architectures, components and their ingredients at various levels of abstraction is a major contribution of its own.

10.2 Directions for Future Work

Elaborating a database construction system like FRAMBOISE is a complex piece of work that requires investigations in various directions. Due to the broadness of the topic, several issues could not be addressed in this thesis and a few restrictive assumptions had to be made.

A first restriction in this thesis has been the focus on *centralized* ADBMSs. This is justified in order to solve simpler problems first. However, a construction system intended for practical applications has to support distributed rule execution as well.

Aspects of concurrency control and recovery are treated in accordance with the functionality specified for the Java Transaction API™ (JTA) [Jav01]. Hence, these aspects are treated reasonably by FRAMBOISE as long as the triggering events are simple (database) events and the actions of the rules have no side effects that are visible outside the database. The complex rule execution models, in particular when parallel rule execution takes place, require more sophisticated transaction models (e.g., nested transactions) than the flat transactions covered by the JTA specification. Even though concurrency control and recovery in ADBMS is not yet fully understood, it might be highly interesting to enhance FRAMBOISE with more sophisticated mechanisms.

FRAMBOISE is conceived as a second-order component framework [Szy97] (cf. Sec. 4.1.3) whereby components form the first tier and skeleton(s) of ECA Systems the second. It would be highly interesting to pursue that approach and to investigate in a third-order component framework that uses FRAMBOISE as one building block. The interoperation design provided by the various connectors that relate an ECA System with the DBMS and external services are a adequate first step in that direction, but further work is still necessary

A final subject for future work is the completion of the construction support environment. A complete implementation is beyond a single thesis, since such an implementation needs many person years of work. Once such a supported system is fully implemented, more experience and quantitative results can be obtained. It were equally feasible to realize more comprehensive active database applications. This application experience, however, is needed to underpin research activities concerning the applicability of active database mechanisms (e.g., the development of tools to design and the maintenance of large rulebases). Furthermore, the provision of credible application examples fosters the application of active database technology.

Appendix A

The Architecture Definition Language WRIGHT

This appendix gives an overview over the principal features of the architecture description language WRIGHT [AG94, AG97, All97].

A.1 The Structure of WRIGHT

Wright 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.

A.1.1 Components

A component description consists of the *computation* part and a number of *ports* that represent the interface of the component. Each port defines a set of interactions in which the respective component may participate. For example, a component representing a (database-) server might have two ports, one to respond to client's queries and another that a database administrator would use to supervise the DBMS (cf. Fig. A.1). The computation part describes what the component actually does. The computation carries out the operations defined for the ports and shows how they are tied together to form a coherent whole. A port specification indicates two aspects of a component. First, it indicates some aspects of the components behaviour. In this view, the ports specification indicates the properties that the component must have if it is viewed through the lense of that particular port. Thus, the port becomes a *partial specification* of the respective component. The computation gives a more complete specification of what is really done.

Component DBServer =
  Port Session [*Session protocol*]
  Port Admin [*Admin protocol*]
  Computation [*Server specification*]

Figure A.1: The Structure of a Component Description

A.1.2 Connectors

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 (cf. Fig. A.2). A role is a partial specification of a connector corresponding to the ports of a component, whereas the glue represents the full (i.e., to the extent required at the architecture level) behavioral specification.

Connector C-S-connector =
  Role Client [*client protocol*]
  Role Server [*server protocol*]
  Glue [*glue protocol*]

Figure A.2: The Structure of a Connector Description

A.1.3 Configurations

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 and connector instances combined via connectors as shown in Fig. A.3. A configuration is completed by describing the *attachments*. The attachments define the topology of the configuration by specifying which components participate in which interactions. This is done by associating a component's port with a connector's role.

The attachment declarations bring together each of the elements of an architectural description. The component carries out a computation, part of which is a particular interaction specified by a port. That port is attached to a role that indicates what rules the port must follow in order to be a valid participant in the interaction specified by

Configuration ClientServer
    Component DBServer =
        Port Session [*Session protocol*]
        Port Admin [*Admin protocol*]
        Computation [*Server specification*]
    Component Client =
        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, Administrator: Client
    cs1, cs2: C-S-connector
  Attachments
    DBMS.Session **as** cs1.Server
    DBMS.Admin **as** cs2.Server
    Application.DBRequest **as** cs1.Client
    Administrator.DBRequest **as** cs2.Client
End ClientServer.

Figure A.3: A Simple Client-Server Configuration

the connector. The glue of the connectors then determine how the computations are combined to form a single, larger computation.

Finally, configurations are applied to specify *hierarchical* descriptions where components or connectors are in turn composed of architectural subsystems. In this case, the computation of a component or the glue of a connector are represented by an architectural description itself. Hence the architectural subsystem is described as a configuration in the same way as indicated above. In addition, however, for a component the nested description has an associated set of bindings, which define how the unattached port names on the inside are associated with the port names (correspondingly for connectors: role names on the inside are identified with the role names on the outside).

A.1.4 Formalizing Architecture Styles in WRIGHT

WRIGHT supports the formal description of architectural styles (cf. Sec. 4.2.2) by means of so-called *style definitions*. A style definition basically defines a set of properties that are shared by the configurations that are members of this style. These properties can be expressed by means of so-called interface definitions, parameterizable connector and component definitions as well as specific style constraints.

Interface Types

An architectural style can restrict the interface of its architectural elements to specific properties. In order to describe these restrictions and to simplify definition, a WRIGHT description can introduce so-called *interface types*. They can be used either as the port of a component or as the role of a connector. In the latter case, the interface type represents a constraint on the port interfaces that may be used in this role. The following example specifies the "call and return" behavior of a procedure call.

Interface Type TProcedureCall = $\overline{call} \rightarrow return?result \sqcap \S$

In principle this interface definition means that the caller initiates the sequence (indicated by means of the overbar) and waits until the callee returns a result. Alternatively a procedure call might not be invoked at all and terminates therefore with an "empty operation" expressed by means of the symbol $\S$. Note that the formalism to express this behavior is presented in greater detail in Section A.2.

Parameterization

In order to expand a description of systems to families of systems, WRIGHT allows to parameterize descriptions i.e., a type description is permitted to leave "holes" in the description to be filled when the type is instantiated. WRIGHT permits any part of the description of a type to be replaced with a *placeholder*. So the type of a role, a

computation, the name of an interface etc. are all a parameterizable. In the following example, the computation part of a component is parameterized, whereas the port definitions are specified by means of an interface type.

Component SingleSessionDBServer (C: Computation) =
Port Session = TProcedureCall
Port Admin = TProcedureCall
Computation = C

Furthermore, component and connector descriptions can be parameterized by *number*, i.e., by indicating a range of integers. Thus, the number of parameters can be used to control the number of particular kinds of ports or roles that can appear. A port or role description that can have multiple copies is indicated by specifying a range of integers as a subscript to its name, as shown in the following example that allows a database server component to have multiple identical session ports.

Component MultipleSessionDBServer (nSession: $1 \ldots$) =
Port $\text{Session}_{1 \ldots nSources} = \overline{open} \rightarrow Wait4Event$
Port Admin = TProcedureCall
Computation = $ControlInfo.open \rightarrow$
$\forall\, i \in 1 \ldots nSources \bullet \overline{Session_i.open} \rightarrow \ldots$

Note that the parameter number of a component or a connector is set at instantiation time and cannot be changed during execution time. Dynamic architectures (i.e., those in which components appear or disappear) are either modeled by including all potential elements in a configuration or by describing each configuration as a different architecture [All97].

Constraints

Finally, a style definition may include explicit constraints which are based on first-order predicate logic. The constraints refer to the following sets and operators:

- *Components:* The set of components in the configuration.
- *Connectors:* The set of connectors in the configuration.
- *Attachments:* The set of attachments in the configuration. Each attachment is represented as a pair of pairs *((comp,port), (Conn, Role))*.
- *Name(e):* The name of element *e*, where *e* is a component, connector, port or role.
- *Type(e):* The type of element *e*.

- *Ports(c):* The set of ports of component *c*.
- *Computation(c):* The computation of component *c*.
- *Roles(c)*: The set of roles of connector *c*.
- *Glue(c):* The glue of connector *c*.

In addition, any type that has been declared as a part of the style's vocabulary may be referred to by name. An example summarizing a simple style definition is given in Figure A.4.

Style ClientServerStyle
  Interface Type TInvoke = [*Call service and wait for result*]
  Interface Type TExec = [*Execute service and return*]
  Connector ProcedureCall =
    Port Caller = TInvoke
    Port Callee = TExec
    Glue [*glue protocol*]
  Constraints
    $\forall\, c \in \mathit{Connectors} \bullet \mathit{Type}(c) = \mathrm{ProcedureCall} \wedge$
    $\forall\, c \in \mathit{Components};\ p : \mathit{Port} \mid p \in \mathit{Ports}(c) \bullet \mathit{Type}(p) = \mathrm{TInvoke} \vee$
      $\mathit{Type}(p) = \mathrm{TExec}$
End ClientServerStyle.

Figure A.4: A Simple Client-Server Style

A.2 Behaviour Specifications

The behaviour and coordination of components and connectors is specified in WRIGHT using a notation based on the process algebra CSP ("Communicating Sequential Processes"). This formalism 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[1] 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.

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

A.2.1 Processes and Events in CSP

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 manner. The behavior pattern of such an object is described by means of *events* which are abstractions of actions performed in the real world.

Processes are denoted in CSP 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*. The alphabet of a process P is a permanent property of this process and is denoted αP, e.g., $\alpha CONVERSATION = \{talk, listen\}$
There is a distinction between event classes and event occurrences. A specific event may occur any number of times. An event name denotes an event class.

The actual occurrence of each event in the life of an object is regarded as an instantaneous or atomic action without duration. In order to model extended or time-consuming operations, they are represented by a pair of events the first denoting the start of an action and the second denoting its completion.

In many situations it is useful to categorize events into special event groups. Thus there is a special class of events known as *communication* written with an infix dot such as $c.\nu$ where c is the name of the channel on which the communication takes place and ν is the value of the message which is passed over the channel.

A.2.2 Traces

A *trace* of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. It is denoted as a sequence of symbols, separated by commas and enclosed in angular brackets.

$\langle start, finish \rangle$ consists of two events, *start* followed by *finish*

$\langle\rangle$ is the empty sequence containing no events

The actual behaviour of a process is represented by means of a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. This string of symbols – addressed as *trace* – is denoted as a list of the respective symbols which are separated by commas. The list is enclosed in angular brackets. For instance, the trace $\langle start, finish \rangle$ consists of two events, *start* followed by *finish*.

A.2.3 Prefixing

The operator $\rightarrow$ allows to *prefix* processes by events which belong to the alphabet of the respective process. For instance, given an event x and a process P regarding that $x \in \alpha P$ then $x \rightarrow P$ is the process which is willing to communicate first the event x and afterwards behaves like P. Note that a prefixed process is a restriction of the original process permitting only those traces over αP that start with the event x. A prefixed process Q where $Q = (x \rightarrow P)$ recurs to P. The concepts of prefixing and recursion enable the description of processes with a single possible stream of behaviour.

A.2.4 Combining Processes

In order to describe processes whose behaviour is influenced by interaction with their environment (represented by other processes), the language 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, recursion and program relabeling. The basic operators are:

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 favor of P and vice versa. $P \sqcap Q$ is an internal choice between P and Q; the outcome of this choice is nondeterministic.

A.2.5 Refinement

The refinement relationship is a way to compare processes that are not identical. It principally guarantees that one process satisfies all of the properties of another, possibly as well as some other properties of its own. It is defined as follows:

A process $P = (\alpha P, F_P, D_P)$ is a refinement of a process $Q = (\alpha Q, F_Q, D_Q)$ written $Q \subseteq P$ iff $\alpha P = \alpha Q \wedge F_P \subseteq F_Q \wedge D_P \subseteq D_Q$.

In a nutshell, a refinement relation ensures that the traces of P are a subset of Q's traces.

A.2.6 Applying CSP to WRIGHT

CSP is applied in WRIGHT to specify the behaviour of the ports, computations, roles and glues. The latter are basically considered as CSP processes which are defined by means of an expression written in the CSP algebra.

WRIGHT extends CSP in some minor syntactic ways. Thus, it distinguishes between *initiating* and *observing* an event. An event that is initiated by a process is written with an overbar. Moreover, 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, a connector that represents a procedure call could be described as follows:

Connector ProcedureCall =
Role Caller = $\overline{call} \rightarrow return?result \sqcap \S$
Role Definer = $call \rightarrow \overline{return!result} \Box \S$
Glue = $Caller.call \rightarrow \overline{Definer.call} \rightarrow Glue$
$\Box\ Definer.return?result \rightarrow \overline{Caller.return!result} \rightarrow Glue$
$\Box\ \S$

Two fractions take part in this interaction whose behaviour is defined as a CSP process by means of the role "Caller" and "Definer". On the one hand there is a component that invokes the procedure, expressed by the overbar $\overline{call}$. Afterwards it waits for the procedure to return the results. Therefore the event $return?result$ is written without an overbar. Note that the role Caller uses the non-deterministic choice operator $\sqcap$ because the caller decides itself whether a procedure is invoked or not. On the other hand the "Definer" provides the procedure and executes it exclusively on an external invocation. Thus *call* is written without an overbar, contrastingly to $\overline{return!result}$ and the expression uses the deterministic choice operator $\Box$. Since the "Glue" process ties the various roles together it must point out which role's event is indicated in any situation. This is achieved by rewriting the role's events in the Glue expression as communication events whereby the respective role is considered as the channel.

The behaviour of the components is specified correspondingly i.e., ports are treated alike the roles and the computation process alike the glue.

A.3 Validating Architectural Descriptions

Architectural specifications written in WRIGHT enable a designer to check a system for consistency and completeness. This analysis is conducted according to a series of eleven tests that are described informally in following paragraphs. The complete formal description can be found in [All97]. Note that we indicate in parentheses the language construct for which each test applies.

Port-Computation Consistency (Component) A port specification must be a *projection* must be a projection of the Computation of its associated component under the assumption that all other port interfaces are obeyed by the environment.

Connector Deadlock-Free (Connector) This test is applied to detect inconsistencies between participants in an interaction and the coordination of that interaction by the glue of a connector. Basically the combination of the roles and the connector's glue must be deadlock-free.

Roles Deadlock-Free (Role) By requiring that each of the roles in a connector must be deadlock-free, this test avoids that role specifications are internally inconsistent.

Single Initiator (Connector) In order to avoid control conflicts, for every event in a connector type specification exactly one of the roles or the glue must initiate the event.

Initiator Commits (Any Process) This rule ensures that the initiate (expressed by means of an overbar over the event) and the observe notations are used consistently. Thus if a process initiates an event, then it must commit to that single event without any

influence by the environment. For example a process like $Invalid = \overline{e} \rightarrow P \,\Box\, \overline{f} \rightarrow Q$ does not obey this rule.

Parameter Substitution (Instance) An instance declaration of a parameterized type must result in a valid non-parameterized type when the actual parameters are substituted for the formal parameters. Thus inconsistencies like name clashes for syntactic placeholders (e.g., port or role names) are avoided when parameterized instances are declared in a configuration.

Range Check (Instance) A numeric parameter must be no smaller than the lower bound, if declared and no larger than the upper bound, if declared.

Port-Role Compatibility (Attachment) Only compatible ports and roles may be attached to each other. That means that the port must handle all of the observed events that the role specifies, but may possibly handle more. The other way round the port may only initiate events that are in the alphabet of the port, it is, however, not mandatory that a port initiates all events a role could process.

Style Constraints (Configuration) The predicates for a style must be true for a configuration declared to be in that style.

Style Consistency (Style) There must be at least one configuration that satisfies the constraints of a style.

Attachment Completeness (Configuration) If a port or role is unattached then it must not depend on observing particular events and it must not be able to initiate any events. Thus every unattached port or role in a configuration must be compatible with the process that simply halts, i.e, $\S$.

Appendix B

The Syntax of the Specification Language

This Appendix presents the complete syntax of specification language introduced in Section 8.1. The syntax is expressed in the extend Backus Naur Form (EBNF).

```
ECAS ::= 'ECAS' ECASName KnowledgeModel ExecutionModel
                    RuleManagementAspects DBMSAspects.

ECASName ::= Name ':'.

KnowledgeModel ::= 'KNOWLEDGE_MODEL' EventTypes
                      ConditionCharacteristics
                      ActionCharacteristics.


EventTypes ::= 'EVENTS' EventSource {EventSource}.

EventSource ::= (External|Database)|'COMPOSITE')';'.

External ::=  'EXTERNAL' ( 'system'|'clock'|'abstract').

Database ::=  'DATABASE' ('modification'|'method'|
              'transaction'|'error').

ConditionCharacteristics ::= 'CONDITIONS' Coupling
                                    ConditionEvaluation.

ConditionEvaluation ::= 'EVALUATION' ['DBMS']
                                     ['stand-alone']';'.
```

```
ActionCharacteristics ::= 'ACTIONS' Coupling ActionExecution.

ActionExecution ::= 'EXECUTION' ['DBMS'] ['stand-alone']';'.

Coupling ::= TemporalCoupling TransactionCoupling

TemporalCoupling ::= 'COUPLINGS' ['immediate']
                                 ['deferred'] ';'.

TransactionBinding ::= 'BINDINGS' ['current']['detached']';'.

ExecutionModel ::= 'EXECUTION_MODEL' CyclePolicy
                                     [ConflictResolution]
                                     RuleConsumptionPolicy
                                     TerminationPolicy.

CyclePolicy ::= 'CYCLE_MODE' ('recursive' |
                              'non-interruptable')';'.

ConflictResolution ::= 'CONFLICT_RESOLUTION'
                               ('none'| ['absolutpriority']
                               ['relativepriority']
                               ['FIFO'] ['LIFO'] |
                               'all')';'.

RuleConsumptionPolicy ::= 'RULE_CONSUMPTION' 'local'|'global' ';'.

TerminationPolicy ::= 'TERMINATION' ('granted'|'limit')';'.

RuleManagementAspects ::= 'RULE_MANAGEMENT' Adaptability.

Adaptability ::= 'ADAPTABILITY' ('compile-time' | 'runtime')';'.

DBMSAspects ::= 'DBMS' Datamodel DML_Statements Interpreter.

Datamodel ::= 'DATAMODEL' ('relational' | 'objectoriented')';'.

DML_Statements ::= 'DML_STATEMENTS' ('interruptable' |
                                'non-interruptable')';'.

Interpreter ::= 'INTERPRETER' ('yes'|'no')';'.

Name :: = Letter { Letter | Digit}.
```

```
Letter ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
           'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'|
           'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'J'|'K'|'L'|'M'|'N'|
           'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'.

Digit ::=  '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
```

Appendix C

The FRAMBOISE Prototype

The FRAMBOISE Component Framework has been implemented as an experimental prototype [Flo84] to *validate* the concepts elaborated in this thesis. The prototype comprehends basically the class framework to develop components i.e., the FRAMBOISE Development Framework (FDF) as presented in Chapter 7.4, components to build representative ECA Systems as well as a set of tools to to assemble ECA Systems according to Fig. 8.1.

This appendix presents the FRAMBOISE prototype in order to illustrate how the construction system is applied by an ADBI. The next Section sketches in which form the prototype is obtained by an ADBI. Subsequently Sections C.2 until C.3 describe how an ECAS is formed out of prefabricated components. Finally, Section C.4 discusses the facilities to define and edit the rulebase of an ECA System.

C.1 Packaging and Installation

The FRAMBOISE Prototype relies on the JavaBean™ Technology [Javb] and has been implemented according to the principles discussed in Chapter 7. Thus *all* elements of the prototype – including the development tools – are basically implemented as JavaBeans.

The Java classes underlying the various JavaBeans are packaged in *Java Archives*, i.e., so-called *JAR Files* that are part of the standard Java infrastructure. A JAR is a compressed file that allows principally to archive arbitrary files, but its main purpose is to package related class files, serialized[1] Java Beans and other resources. This scheme allows multiple beans to be packaged in a single JAR file, providing a convenient way to share common class files.

Thus FRAMBOISE components are delivered as a number of Java Archives and the FRAMBOISE prototype is basically installed by adding them to the so-called class

[1]The so-called *Object Serialization* is a Java mechanism that supports the encoding of objects, and the objects reachable from them, into a stream of bytes as well as the complementary reconstruction of the object graph from the stream. Serialization is among other things used for lightweight persistence.

path[2]. Upon installation an ADBI can construct ECAS as discussed in the subsequent sections.

C.2 Specification of the Functionality of the ECA System

Constructing an ECAS starts with specifying it by means of the specification language presented in Section 8.1. The FRAMBOISE prototype comprises a specification language compiler to process ECAS specifications. The specification compiler generates an initial version of a so-called *configuration file* that basically describes which ingredients (e.g., JavaBeans, resource files, database schemas etc.) form the prospective ECA System.

In order to assist the ADBI in specifying the ECAS, the workbench[3] offers a specification editor as depicted in Figure C.1. The sliders on top enable the ADBI to select

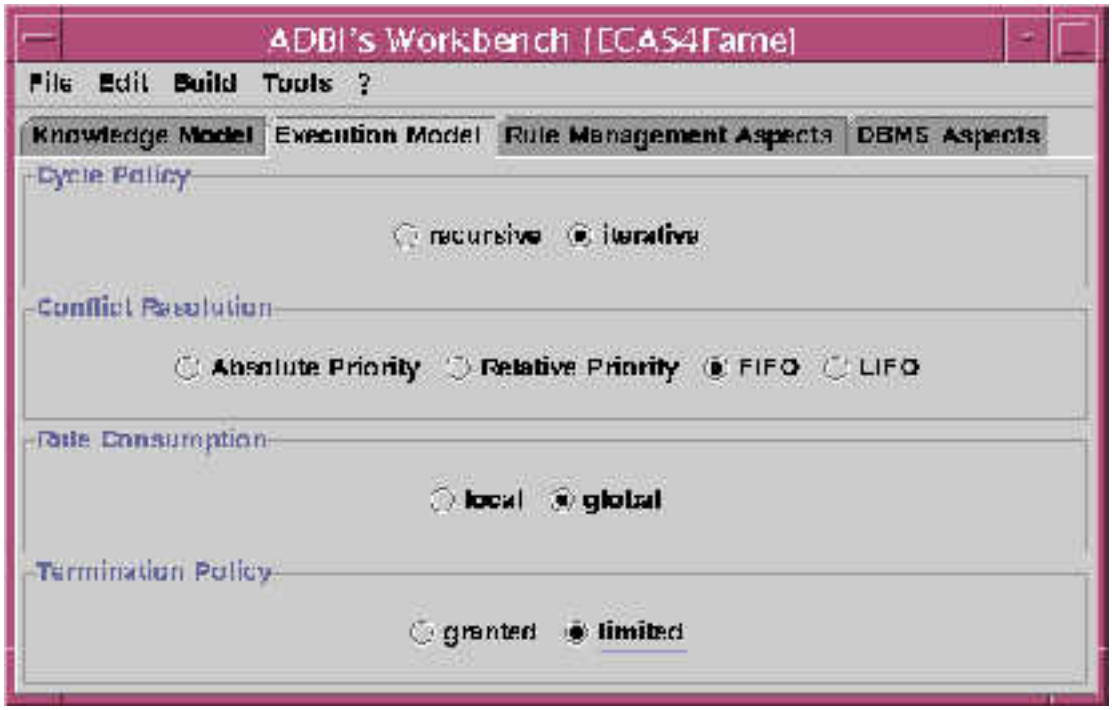

Figure C.1: The FRAMBOISE Specification Editor

[2]This Java specific term refers to an environment variable that tells Java tools and applications where to find third-party and user-defined classes, i.e., classes that are not Java extensions or part of the Java platform.

[3]All facilities to construct ECA Systems are actually bundled into one tool that has been named as *ADBI's Workbench*.

the various aspects of the ECAS specification in order to determine their facets as shown in this example for the rule execution model. The specification editor gives the ADBI opportunity to check the specification for inconsistencies (e.g., recursive rule cascades even though action execution cannot be interrupted to perform the recursion). It is furthermore feasible to invoke the specification compiler in order to generate a configuration file. The latter is subsequently processed by the configuration manager.

C.3 Configuring an ECAS

The configuration manager enables an ADBI to browse the component repository and to connect components into a coherent ECAS. The user interface of the configuration manager is shown in Figure C.2. On the left hand the structure of the ECASs under

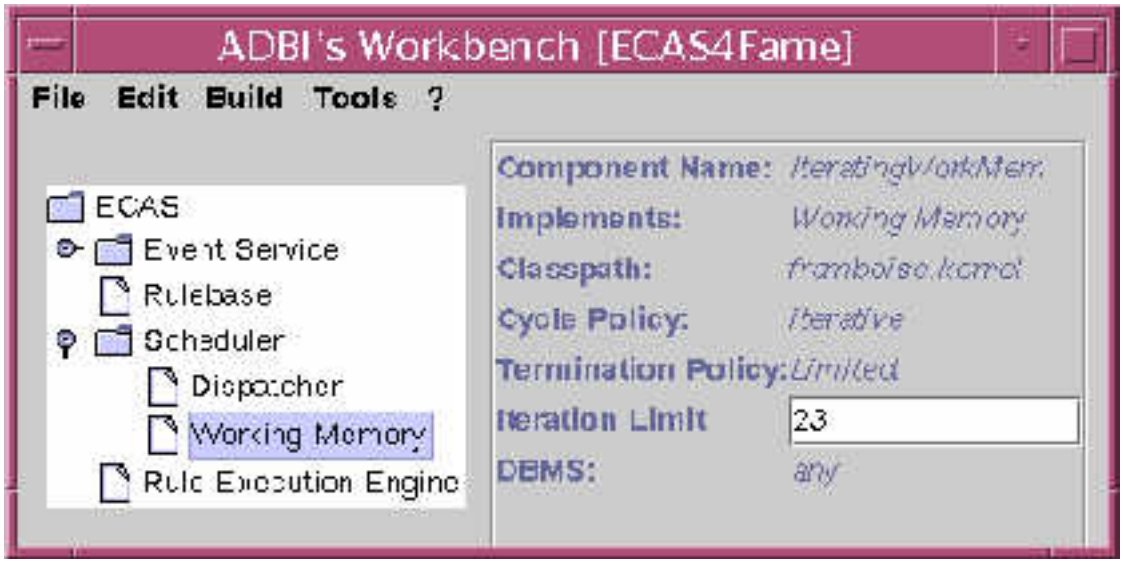

Figure C.2: The FRAMBOISE Component Manager

construction is displayed in a tree-like manner. A node in this tree represents a specific component of the ECAS. By unfolding such a node the subcomponents of the component represented by this node are displayed.

A node that is selected has the information about the respective component displayed on the panel on the right hand. This panel is specific for each component and is built out of specialized classes that belong to the respective Bean. The JavaBeans standard [Javb] establishes various classes and interfaces that help visual design tools to use beans in a design environment. Among other things, there are so-called *customizer classes* that are furnished by a Bean implementor to provide a complete custom GUI for customizing a target Java Bean.

The example in Figure C.2 displays a Bean implementing a working memory. The editor panel on the right side is actually the customizer of this Bean and requires here the ADBI define the limit of cascading rules.

The button “Check”, invokes the configuration manager validate whether all required components are configured properly. Furthermore the the Configuration Manager enables the ADBI to generate an ECAS on account of the actual configuration. Generating an ECAS consists of the following activities:

- Creation and serialization of JavaBean instances.
- Creation the configuration file with the necessary runtime information to initialize the ECA System (e.g., loading the appropriate Bean instances and connecting them to the InfoBus, cf. Sec. 7.3.4) at startup.
- Packaging all building blocks in a ECAS specific JAR.
- Creation of the necessary directory structure.
- Creation of an (empty) rulebase.

An ECA System furnished by the FRAMBOISE Prototype is equally a prototype that runs as multi-threaded application in *one* Java virtual machine in order to avoid wicked interprocess communications.

Note that an ADBI eventually has to furnish novel components before an ECAS can be configured. Thus the FDF is a part of the prototype so that the provision of a new component is is performed as described in Chapter 7.

C.4 Define the Rulebase

Once the construction of an ECAS is achieved, one has to define its rulebase. For that purpose an interactive rulebase editor is furnished that provides graphical interfaces for inserting into, deleting from and querying the rulebase. It displays individual rules, events, conditions and actions in different windows and establishes relationships between them. For example consider a rule `AdjustAccounts` that is displayed in a rule editor as shown in Figure C.3. The constituents of this rule, namely an event called `UpdateSalaryAccount`, the condition `oddDistibution` and an action named `adjustAccounts` and their coupling modes are shown in the panel below the list displaying the names of all rule definition in this rulebase. Once again this panel is not a fixed building block of the rulebase editor but it is actually a customizer of the rule definition bean and is therefore dynamically loaded on account of the ECAS’ configuration file. Clicking on the buttons on the left side opens a specific editor of each rule constituent. For example the button `E` opens the event editor for the event as shown in Figure C.4. The upper part of the event editor is a list of all events in this rulebase accompanied by a list of all rulenames in which the selected event appears. Those

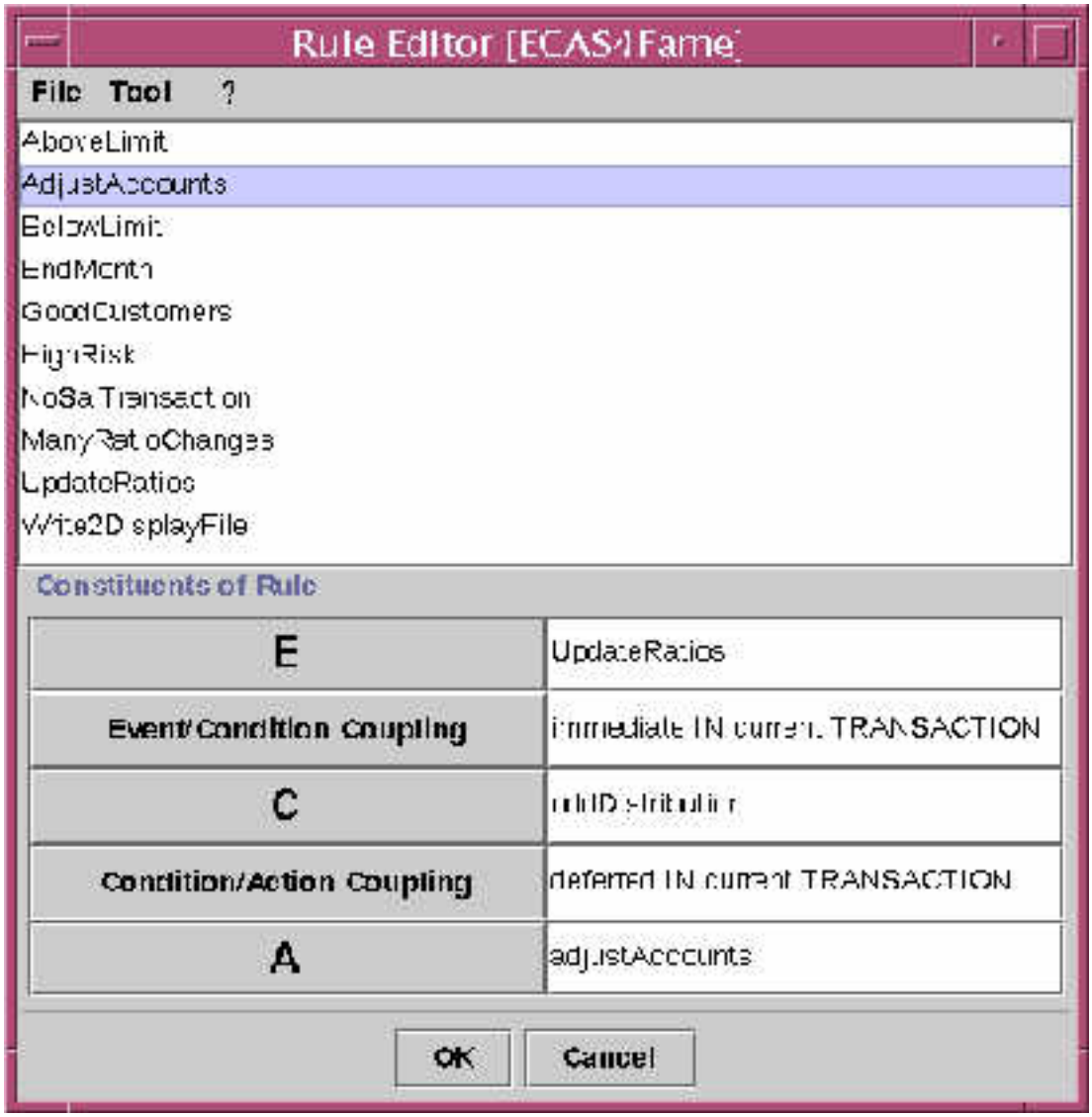

Figure C.3: The FRAMBOISE Rule Editor

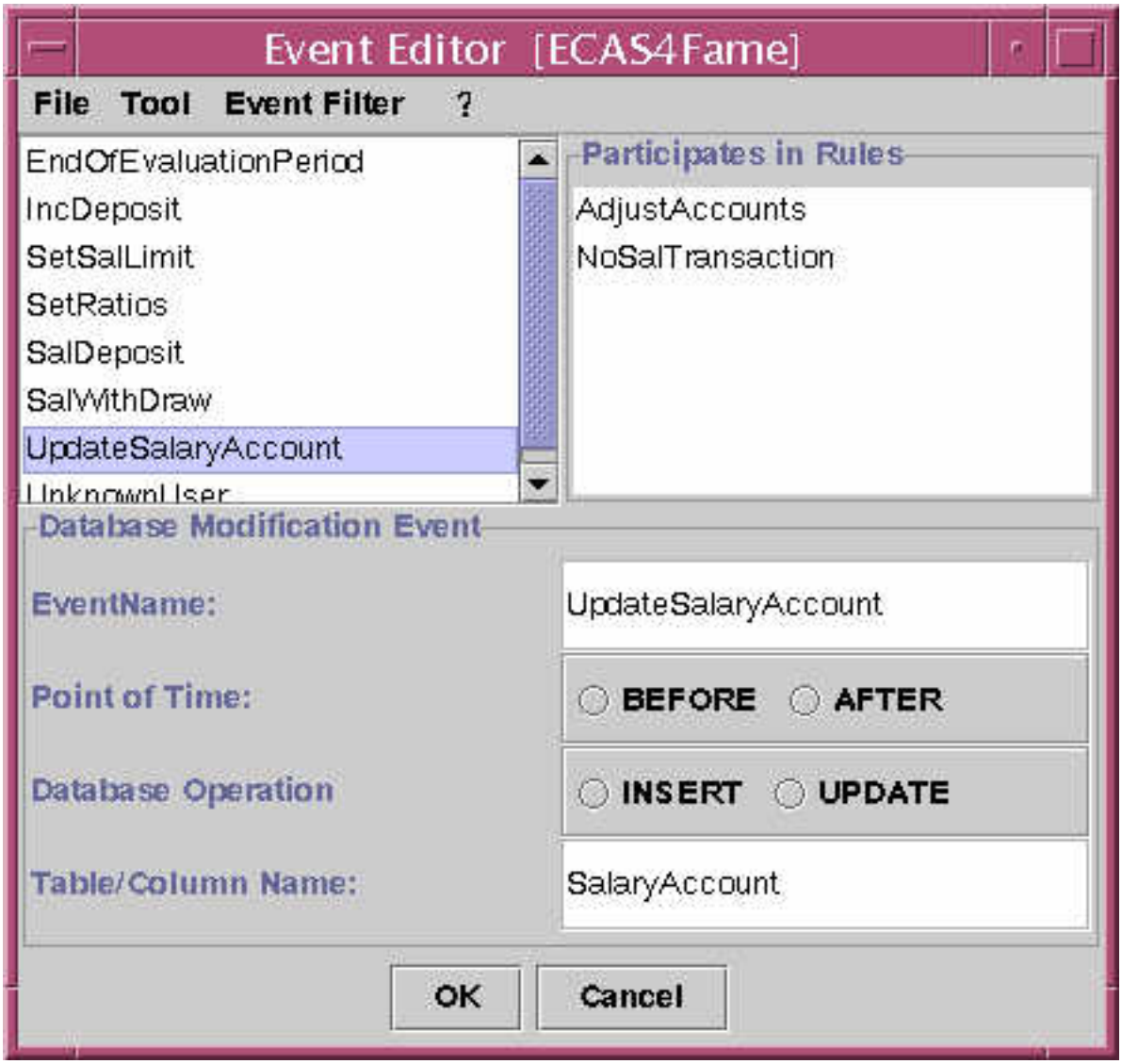

Figure C.4: The FRAMBOISE Event Editor

visual elements are applicable for all event definitions ,and are therefore a fixed part of the event editor.

Below is the customizer for the respective event definition. In the example the event `UpdateSalaryAccount` applies for relational DBMSs, i.e., it is signalled after an update of a table named `SalatryAccount`.

In order to define a new event, a list of all event types that are applicable for this ECAS is displayed. The ADBI determines the event type by selecting one element of this list. Afterwards this panel is replaced by the appropriate customizer.

Conditions and Actions are equally implemented as Java Beans thereby relying on the design pattern [GHJV95] *command*. In a nutshell, beans representing actions implement a common interface called `IAction` that defines a method named `execute()` [4]. A class furnishing a concrete action overwrites this method which is at the proper point of time invoked by the ECAS.

A concrete action class can then be loaded dynamically as any Java class by the Java virtual machine at runtime from the ECAS.

[4] Conditions obey a similar convention by implementing a common interface that define a method returning a boolean value.

Appendix D

Acronyms

ADBS	Active Database System
ADBI	Active Database Implementor
API	Application Programming Interface
CDBMS	Component Database Management System
CORBA	Common Object Request Broker Architecture
CSP	Communicating Sequential Processes
COM	Component Object Model
COP	Component Oriented Programming
CRUD	Create Read Update Delete
DBMS	Database Management System
DML	Data Manipulation Language
ECA	Event Condition Action
FRAMBOISE	A Framework Using Object-Oriented Technology for Supplying Active Mechanisms
IDL	Interface Definition Language
J2EE	Java 2 Platform Enterprise Edition
JTA	Java Transaction API
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
RDL	Rule Definition Language
UML	Unified Modeling Language
XP	Extreme Programming

Curriculum Vitae

Last name:	Fritschi
First name:	Hans
Date of birth:	July 19th 1961 in Romanshorn
Place of origin:	Flaach, ZH

Education

1968 – 1973	Primarschule Schaffhausen and Zurich
1973 – 1976	Sekundarschule, Zurich
1977 – 1981	Gymnasium Juventus
1982 – 1982	Eidg. Matura
1983 – 1985	Military Service
1983 – 1986	Studies in Mathematics, ETH Zurich
1987 – 1994	Studies in Computer Science and Business Administration, University of Zurich, graduation in December 1994
1995 – 2000	Ph.D. student, University of Zurich

Professional Experience

1987 – 1993	Elektrowatt Ingenieurunternehmungen AG Zurich, Part time employment as Software Developer
1993 – 1994	Erich Gueng Beratungen AG, Dübendorf, Part time employment as Software Developer
1995 – 2000	Research assistant at the Department of Information Technology, University of Zurich
2000 – 2002	Tarsec Inc Zurich Full time employment as Senior Software Architect
2002 –	Credit Suisse Financial Services Zurich Full time employment as Lead Engineer

Bibliography

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[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.

[ASU86] A. Aho, R. Sethi, and J. Ullmann. *Compilers, Principles Techniques and Tools*. Addison Wesley, 1986.

[BAMS93] K. Beck B. Anderson M. Shaw, L. Best. Software architecture: The next step for object technology. In ACM Press, editor, *OOPSLA93, ACM SIGPLAN Notices*, volume 28, pages 356–359, Reading, MA, 1993.

[BB91] B. H. Barnes and T. E. Bollinger. Making reuse cost effective. *IEEE Software*, pages 13–24, January 1991.

[BBG+88] D. Batory, J. R. Barnett, J. F. Garza, K. P. Smith K. Tsuda, B. C. Twichel, and T. E. WiseTlchew. Genesis an extensible database management system. *IEEE Transaction on Software Engineering*, 14:1711–1729, November 1988.

[BBK+97] F. Bronsard, D. Bryan, W. Kozaczynski, E.Liongosari, J. Q. Ling, A. Olafsson, and J. W. Wetterstrand. Toward software plug and play. In *ACM Software Engineering Notes*, volume 22, pages 19–29, 1997.

[BCK98] L. Bass, P. Clements, and R. Katzman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BDD+95] O. Boulcema, J. Dalrymple, M. Doherty, J.C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating active and multi database services into an osa-compliant introperability toolkit. *The Collected Arcadia Papers, Second Edition*, 1995.

[Bec99] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70 – 77, October 1999.

[Bec00] K. Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[BFL+97] M. Bouzeghoub, F. Fabret, F. Llirbat, M. Matulovic, and E. Simon. Active-design: A generic toolkit for deriving specific rule exmodels. In A. Geppert and M. Berndtsson, editors, *Rules in Database systems*, pages 197 – 211, Skovde, Sweden, June 1997. Springer-Verlag.

[BL93] M. Berndtsson and B. Lings. On developing reactive object-oriented databases. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Database Research*, January 1993.

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

[Boe88] B. W. Boehm. A spiral model of of software development and enhancement. *IEEE Software*, 25(5):61–72, May 1988.

[Boo87] G. Booch. *Software Components with Ada: Structures Tools and Subsystems*. The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA, 1987.

[Boo94] G. Booch. *Object-Oriented Analysis and Design*. The Benjamin/Cummings Publishing Company, Inc, 2 edition, 1994.

[BP95] A. Biliris and E. Panagos. A high performance configural storage manager. In *Proc. 11th Intl. Conference on Data Engineering*, Tapei, Taiwan, 1995.

[BRJ98a] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley Object Technology Series. Addison Wesley, 1998.

[BRJ98b] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Object Technology Series. Addison Wesley, 1998.

[Bro84] L. Brodi. *Thinking FORTH*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[Bro87] F. Brooks. No silver bullets – essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.

[Bro95] F. Brooks. *The Mythical Man Month – Essays on Software Engineering*. Addison Wesley, Reading, MA, 20th anniversary edition, 1995.

[Buc98] A. P. Buchmann. Architecture of active database systems. In *[Pat98]*, pages 29–48. Springer, 1998.

[BW98] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–47, September 1998.

[BZBW95] A.P. Buchmann, J. Zimmermann, J.A. Blakeley, and D.L. Wells. Building an integrated active oodbms: Requirements, architecture and design decisions. In *Proc. of the 11nth intl Conference on Data Engineering*, pages 117–128, Taipei, Taiwan, 1995. IEEE Computer Society Press.

[CBB+88] S. Chakravarty, B. Blaustein, A. Buchmann, M.J. Carey, U. Dayal, M. Hsu, R. Jauhari, R. Ladin, M. McCarthy, and D. Rosenthal. The hipac project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1), March 1988.

[CC96] C. Collet and T. Coupaye. Composite events in NAOS. In *Proc. of the 7nth Intl Conf. on Database and Expert Systems, DEXA 96*, Zurich, Switzerland, September 1996.

[CCS94] C. Collet, T. Coupaye, and T. Svensen. Efficient and modular reactive capabilities in an object-oriented database system. In *Proc. of the 20 Intl. Conference on Very Large Databases*, pages 132 –143, September 1994.

[CDG+90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The exodus extensible dbms project: An overview. Technical report, University of Wisconsin-Madison, 1990.

[CFPT96] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. *Active Database Systems: Triggers and Rule for Advanced Database Processing*, chapter Active Rule Management in Chimera, pages 152–176. Data Managment Systems. Morgan Kaufmann, 1996.

[CH90] M. Carey and L. Haas. Extensible database management systems. *SIGMOD Record*, 19(4), 1990.

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From custom applications to domain specific frameworks. *Communications of the ACM*, 40(10):70–77, October 1997.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. of the 20th Intl. Conference on Very Large Databases*, Santiago, Chile, September 1994.

[CKTB95] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. Rule integration in an oodbms: Architecture and implementation. In *Proc. of the 11nth intl Conference on Data Engineering*, pages 117–128, Taipei, Taiwan, March 1995. IEEE Computer Society Press.

[Cle88] J. C. Cleaveland. Building application generators. *IEEE Software*, 4(5):25–33, July 1988.

[Clo] *Cloudscape - Product Documentation*. www.cloudscape.com.

[CM90] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software*, 7(6), November 1990.

[CP95] S. Cotter and M. Potel. *Inside Taligent Technology*. Addison-Wesley, 1995.

[Day88] U. Dayal. Active database management systems. In *Proceedings of the 3rd Intl. Conference n Data and Knowledge Bases*, Jerusalem, June 1988.

[DG00] K. R. Dittrich and A. Geppert, editors. *Component Database Systems*, chapter 1. Morgan Kaufmann, 2000.

[DGG95] K.R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto. In T. Sellis, editor, *Proc. of 2nd Intl Workshop on Rules in Database Systems*, Athens, Greece, September 1995. Springer.

[Dia98] O. Diaz. Tool support. In *[Pat98]*, pages 127–145. Springer, 1998.

[Dij68] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341 – 346, May 1968.

[DPP91] O. Diaz, N. Paton, and P.Gray. Rule management in object-oriented databases: A uniform approach. In *Proc. of the 17nth Intl. Conf. on very Large Data Bases*, Barcelona, Spain, 1991.

[DW99] D. D'Souza and A. Wills. *Objects, Components and Frameworks in UML*. Addison Wesley, 1999.

[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.

[FI94] W. B. Frakes and S. Isoda. Success factors of systematical reuse. *IEEE Software*, 11(5), September 1994.

[Flo84] C. Floyd. A systematic look at prototyping. In *Approaches to Prototyping*. Springer, Heidelberg, 1984.

[FM96] J. Feiler and A. Meadow. *Essential OpenDoc*. Addison Wesley, Reading, Mass., 1996.

[FP94] W. B. Frakes and Th. P. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, August 1994.

[Fra94] W. B. Frakes, editor. *3rd International Conference on Software Reuse*. IEEE Computer Society Press, November 1994.

[FSJ99] M. Fayad, D. C. Schmidt, and R. Johnson, editors. *Building Application Frameworks*. John Wiley & Sons, 1999.

[FT95] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20(4):414 – 471, December 1995.

[Gat94] S. Gatziu. *Events in an Active Object-Oriented Database System*. PhD thesis, University of Zurich, 1994.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1987.

[GD94a] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In N. W Paton and H. W. Williams, editors, *Rules in Database Systems*. Springer, 1994.

[GD94b] S. Gatziu and K.R. Dittrich. Detecting composite events in an active database system using petri nets. In J. Widom and S. Chakravarty, editors, *Proc. 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, Houston TX, 1994. IEEE Computer Society Press.

[GD94c] 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.

[Gep94] A. Geppert. *Methodical Construction of Database Management Systems*. PhD thesis, University of Zurich, 1994.

[GGD95a] S. Gatziu, A. Geppert, and K. R. Dittrich. The SAMOS active dbms prototype. In *Proceedings of the ACM SIGMOD Conference*, page 480, 1995.

[GGD+95b] A. Geppert, S. Gatziu, K. R. Dittrich, H. Fritschi, and A. Vaduva. Architecture and implementation of the active object-oriented database management system samos. Technical Report 95.29, Institut fuer Informatik, Universitaet Zuerich, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley professional computing series. Addison Wesley, 1st edition, 1995.

[GJ91] N.H. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of the 17nth Intl. Conf. on Very Large Data Bases*, Barcelona, September 1991.

[GJS92] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases. In *Proc. of the 18th Intl. Conference on Very Large Databases*, Vancouver, August 1992.

[GJS96] A. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading MA, 1996.

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

[Gro95] ObjectManagement Group, editor. *CORBA Services: Common Object Services Specification*, volume 1. John Wiley & Sons, NJ, 1995.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[GSD97] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: a construction approach for database management systems based on reuse. Technical Report 96.05, Dept. of Computer Science, University of Zurich, 1997.

[Hae87] Th. Haerder. *Datenbank-Handbuch*, chapter Realisierung von operationalen Schnittstellen, pages 167 – 342. Springer, 1987. In German.

[Han96] E. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), February 1996.

[HC91] J. W. Hooper and R. O. Chester. *Software Reuse: Guidelines and Methods*. Plenum Press, New York, 1991.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohmann, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst Mid-Flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[HK97] E. N. Hanson and S. Koshla. An introduction to the triggerman asynchronous trigger processor. In A. Geppert and M. Berndtsson, editors, *Rules in Database systems*, pages 51–67, Skovde, Sweden, June 1997. Springer-Verlag.

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

[HR99] Th. Haerder and E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 1999. In German.

[IBM95] IBM Corp. *DB2 relational Extenders*, April 1995.

[Inf98] Informix. Developing datablade modules for Informix dynamic server with universal data option. White Paper, Informix Corp., 1998.

[Jac93] I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA, revised printing edition, 1993.

[Jas94] H. Jasper. Active databases for active repositories. In *Proc. 10. Intl Conference on Data Engineering*, pages 375–384, Houston, USA, 1994.

[Java] JavaSoft. *Java 2 Platform Enterprise Edition*. Sun Microsytems, Inc., http://java.sun.com/j2ee/download.html. Version 1.21.

[Javb] JavaSoft. *JavaBeans API Specification*. Sun Microsytems, Inc., http://java.sun.com/beans. Version 1.01.

[Jav01] JavaSoft. *Java Transaction API*. Sun Microsytems, Inc., http://java.sun.com/products/jta, 2001. Version 1.0.1.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *The Journal of Object-Oriented Programming*, 1(2):22 – 35, 1988.

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

[Joh97] R. E. Johnson. Components, frameworks, patterns. In *ACM Software Engineering Notes*, volume 22, pages 10–17, 1997. extend abstract.

[Kan87] K. C. Kang. A reuse-based software development methodology. In *Workshop on Software Reuse*, Boulder, CO, 1987.

[Kar95] E.A. Karlsson, editor. *Software Reuse : A Holistic Approach*. John Wiley & Sons, 1995.

[KdRB91] G. Kiczales, J. de Riviere, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KDS98] A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, commercial experience and beyond. In *[Pat98]*, pages 366–404. Springer, 1998.

[KGvBF98] A. Koschel, S. Gatziu, G. v. Bültzingsloewen, and H. Fritschi. Applying the unbundling process on active database systems. In *Intl. Workshop on Issues and Applications of Database Technoilogy (IADT)*, Berlin, July 1998.

[KHS94] G. Knolmayer, H. Herbst, and M. Schlesinger. Enforcing business rules by the application of trigger concepts. In *Proc. of the Priority Programme Informatics Research, Information Conference*. Swiss National Science Foundation, November 1994.

[KRT89] S. Katz, C. A. Richter, and K. The. Paris: A system for reusing partially interpreted schemas. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume I. ACM Press, 1989.

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

[LH89] M. D. Lubars and M. T. Harandi. Knowledge-based software design using design schemas. In *Proceedings of the 9th Intl. Conference on Software Engineering*, pages 252–262, March 1989.

[Mar94] J. J. Marciniak, editor. *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[Mau00] P. Maurer. Components: What if they gave a revolution and nobody came. *IEEE Computer*, 33(6):28–34, June 2000.

[MBH+86] F. Maryanski, J. Bedell, S. Hoelscher, S. Hong, L. McDonald, J. Peckham, and D. Stock. The data model compiler: A toll for generating object-oriented database systems. In *Proc. of the 1986 Intl. Workshop on Object-Oriented Database Systems*. IEEE Computer Science Press, 1986.

[MD89] D. R. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proceedings of the 1989 ACM SIGMOD International Conference of the Management of Data*. ACM SIGMOD, ACM SIGMOD, 1989.

[Med96] N. Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report Irvine, California 92697-3425, Dept. of Information and Computer Science, University of California, Irvine, 1996.

[Mey92a] B. Meyer. Applying ' design by contract'. *IEEE Computer*, 25(10):40 – 51, 1992.

[Mey92b] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mic] Oberon Microsystems. Jbed whitepaper: Component software and real-time computing. http://www.oberon.ch.

[MKMG97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1), January 1997.

[Moo94] J. Moore. Debate on software reuse libraries. In *[Fra94]*, pages 203–204, 1994.

[NL97] O. Nierstrasz and M. Lumpe. Komponenten, Komponentenframeworks und Gluing. *Theorie und Praxis der Wirtschaftsinformatik*, (197):9–23, September 1997. in german.

[NR68] P. Naur and B. Randell, editors. *NATO Conference on Software Engineering*, Garmisch, October 1968. NATO Science Committee, Brussels. Published as Book in 1976.

[NT95] O. Nierstrasz and D. Tsichritzis, editors. *Object Oriented Software Composition*. Prentice Hall, London, 1995.

[obj] objectmentor.com, http://www.junit.org. *Testing Ressources for Extreme Programming*.

[Obj93] Object Design. *ObjectStore - Manuals for Release 4.0 for SOLARIS*, 1993.

[Obj97] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.0 edition, 1997. formal document 97-02-25.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley, 1996.

[Ora92] Oracle Coorporation. *Oracle - Manual for Release 7.0*, 1992.

[ora97] Managing text with oracle8 context cartridge, June 1997. Technical White Paper.

[Ora99] Oracle. All your data: The oracle extensibility architecture. Technical White Paper, 1999.

[Pat98] N. W. Paton, editor. *Active Rules in Database Systems*. Springer, 1998.

[PD85] R. Prieto-Díaz. *A Software Classification Scheme*. PhD thesis, University of California, Irvine, 1985.

[PD93] R. Prieto-Díaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.

[PD94] R. Prieto-Díaz. Historical overview. In *[SPDM94]*, chapter 1, pages 1–16. Ellis Horwood, New York, 1994.

[PD98] N. Paton and O. Diaz, editors. *Active Rules for Databases*, chapter Introduction. Springer Verlag, April 1998.

[PDA91] R. Prieto-Diaz and G. Arango, editors. *Domain Analysis and Software Systems*. IEEE Computer Society Press, 1991.

[PDF87] R. Prieto-Díaz and P. Freemann. Classifying software for reusability. *IEEE Software*, pages 6–16, January 1987.

[PDW+93] N. W. Paton, O. Diaz, M. H. Williams, J.Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N. Paton and M. Williams, editors, *Proc. of the 1st International Workshop on Rules in Database Systems*, Edinburgh, 1993.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Reading, Mass., 1994.

[Pre97] W. Pree. *Komponentbasierte Softwareentwicklung mit Frameworks*. dpunkt, Heidelberg, Germany, 1997. in German.

[PSS+87] H. Paul, H. Schek, M. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the darmstadt database kernel system. In *SIGMOD Conference*, pages 196–207, 1987.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[PY93] J. S. Poulin and K. P. Yglesias. Experiences with a faceted classification schema in a large reusable software library. In *Seventeenth Annual International Computer Software and Applications Conference*, pages 90–99q, Phoenix, AZ, November 1993.

[Ran57] S. R. Ranghanathan. *Prolegomena to Library Classification*. The Garden City Press Ltd., 1957.

[Ray01] E. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O' Reilly, 2001.

[Rin97] D. C. Rine. Supporting reuse with object technology. *IEEE Computer*, 30(10):43–45, October 1997.

[Rog97] D. Rogers. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[Sam97] J. Sametinger. *Software Egineering with Reusable Components*. Springer, 1997.

[SB99a] M. Stonebraker and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave*, chapter 11 – Implementation of Rule Systems, pages 151–161. Database Management Systems. Morgan Kaufman, 2nd edition, 1999.

[SB99b] M. Stonebraker and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave*. Database Management Systems. Morgan Kaufman, 2nd edition, 1999.

[SBF96] S. Sparks, K. Benner, and C. Farris. Managing object-oriented framework reuse. In M. E. Fayad and M. Cline, editors, *IEEE Computer Theme Issue on Managing Object-Oriented Software Development*. 1996.

[Sch97] H. A. Schmidt. Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51, October 1997.

[Ses98] P. Seshadri. Enhanced abstract datatypes in object-relational databases. *VLDB Journal*, 7(3):130–140, 1998.

[Sha96] M. Shaw. Some patterns for software architechture. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design*, volume 2 of *Second Annual Conference on Pattern Languages of Programming 1995*, pages 255–269. Addison-Wesley, 1996.

[SK91] M. Stonebraker and G. Kemnithz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10), October 1991.

[SKD95] E. Simon and A. Kotz-Dittrich. Promises and realities of active database systems. In P. Gray S. Nishio U. Dayal, editor, *21st Int. Conference on Very Large Databases*, pages 642 – 653. Morgan Kaufman, 1995.

[SLMH96] P. Steyaert, C. Lucas, K. Mens, and K. De Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the OOPSLA*, pages 268–285, October 1996.

[SN92] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, December 1992.

[Som92] I. Sommerville. *Software Engineering*. Addison-Wesley, 4th ed. edition, 1992.

[SP96] C. Szyperski and C. Pfister. Workshop on component-oriented programming. In M. Mülhäuser, editor, *Special Issues in Object-Oriented Programming - ECOOP96*. dpunkt Verlag, Heidelberg, 1996.

[SPDM94] W. Schäffer, R. Prieto-Díaz, and M. Matsamuto, editors. *Software Reusability*. Ellis Horwood, New York, 1994.

[Spe88] H. Spencer. How to steal code. In *Proceedings of the Winter 1988 Usenix Technical Conference*, 1988.

[SQL99] ISO: Database Languages – SQL – Part 2: Foundation. ISO/IEC9075-2:1999, 1999.

[SW94] R. A. Snowdon and R. C. Warboys. An introduction to process-centered environments. In A. Finkelstein an J. Kramer and B. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press Limited, 1994.

[Szy97] C. Szyperski. *Component Software*. Addison-Wesley, 1997.

[Tal95] Inc. Taligent, editor. *The Power of Frameworks*. Addison-Wesley, 1995.

[tas95] CORBAfacilites: Task management common facilities. Technical Report formal/97-06-09, Object Management Group, 1995. V4.0 November 1995.

[TGD97] D. Tombros, A. Geppert, and K. R. Dittrich. Semantics of reactive components in event-driven workflow execution. In *Proc. 9th Intl. Conf. on Advanced Information systems Engineering*, Barcelona, Spain, June 1997.

[Tom99] D. Tombros. *An Event- and Repository-Based Component Framework for Workflow System Architecture*. PhD thesis, University of Zurich, November 1999.

[Vad99] A. Vaduva. *Rule Development for Active Database Systems*. PhD thesis, University of Zurich, 1999.

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

[Vas95] D. Vaskevitch. Very large databases how large? how different? In *Proceedings of the 21st Intl. Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[VK89] D. M. Volpano and R. Kieburtz. The templates approach to software reuse. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume I. ACM Press, 1989.

[WBT92] D. L. Wells, J. A. Blakeley, and J. A. Thompson. Architecture of an open object-oriented database management system. In *Computer*, volume 25, October 1992.

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

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to starburst. In *Proceedings of the 17nth Intl. Conference on Very Large Databases*, pages 275–285, Barcelona, Spain, September 1991.

[Wei96] T. Weik. *Termination and Confluence in an Active Object-Oriented Database*. PhD thesis, Fakultät für Informatik und Automatisierung, TU Ilmenau, 1996. in german.

[Wir99] N. Wirth. Records, objects, classes, modules and components. In Honour of Tony Hoares Retirement, Oxford, 1999.

[WWC92] C. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.

[ZWH95] S. Zweben, B. Weide, and J. Hollingsworth. The effects on layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering*, 21(3):200–208, 1995.