KIDS: Construction of Database Management Systems based on Reuse

Andreas Geppert, Stefan Scherrer, Klaus R. Dittrich

Technical Report 97.01

Institut für Informatik, Universität Zürich
Switzerland

January 9, 1997

Abstract

This paper describes the KIDS[1] approach to database management system (DBMS) construction. KIDS aims at the development of a DBMS-construction approach by defining specification-based approaches strongly relying on software reuse. We propose an architecture model that supports reuse of architectures, but is also able to integrate various DBMS components into a coherent entirety. Second, KIDS defines a construction process model that strives for reuse in all phases. We have identified three important aspects that are initially addressed by the construction approach: the data model aspect, the transaction management aspect, and the integrity management aspect. For each of these aspects, a customized approach how to construct the corresponding subsystem has been developed, again relying on software reuse wherever possible. While these aspects have been considered because of their relevance for any DBMS, the construction method is open for further aspects.

Keywords: Database management systems, architecture of database management systems, software engineering, reuse, specification, data models, transaction management, integrity constraints

1. Kernel-based Implementation of Database management Systems

Table of Contents

1 Introduction and Motivation 3

1.1 KIDS in a Nutshell 5

1.2 Outline 5

2 The Software Engineering Perspective on DBMS-Construction 6

2.1 Software Reuse 6

2.2 Software Architecture 7

2.3 Software Processes 8

2.4 Software Engineering and DBMS-Construction 8

3 The KIDS Architecture Model 9

3.1 Architectural Style and General Decisions 10

3.2 The Broker/Services Model 11

3.3 The Initial Architecture 14

3.4 Example 16

4 The KIDS Construction Process 17

4.1 The KIDS Meta Architecture 17

4.2 Overview of the DBMS-construction Process 19

4.3 DBMS-Analysis 22

4.4 Architecture Selection 23

4.5 Aspect Specification 23

4.6 Architecture Refinement 24

4.7 Subsystem Development 24

4.7.1 Subsystem Design 24

4.7.2 Subsystem Implementation 26

4.7.3 Subsystem Integration 26

4.8 The KIDS Meta Process 27

5 Construction of Selected DBMS Aspects 28

5.1 The Data Model Aspect 28

5.2 The Transaction Model Aspect 31

5.3 The Integrity Maintenance Aspect 35

6 Related Work 38

7 Status and Future Work 41

8 References 41

1 Introduction and Motivation

In the good old days of DBMS-research and development, there was a small set of different DBMS-types (relational, hierarchical, and network). In addition to the three standard data models, further tasks have been “quasi-standardized”, e.g., most systems used two-phase locking to ensure serializability of transactions. However, over the past decade, researchers started to investigate additional functionality in order to support new application areas or to improve support for existing ones. Examples of such new application areas are design applications (CAD, CASE), geographical information systems (e.g., satellite image processing), scientific and statistical data management (e.g., in molecular biology), computer integrated manufacturing (CIM), information retrieval systems, etc. Functionality to be incorporated into DBMSs comprises new data model concepts, advanced transaction models, storage and retrieval of multimedia data, deductive and active rule mechanisms, and many more.

As a consequence, a bulk of new DBMSs with enhanced functionality has been constructed and even more will have to be constructed in the future. The trend to move ever more functionality into the DBMS yields the *problem of DBMS construction*. It is thus a challenging research topic to investigate how to construct DBMSs systematically [28]. Moreover, progress in DBMS-construction is of paramount importance for both, research and industrial practice.

As for a sample scenario, assume that a company wants to offer DBMS-products for a variety of application areas (say, a “plain” object-oriented DBMS, a DBMS-based video-server, a CASE-DBMS, and a real-time DBMS for plant control). Such a company apparently has several options to implement its product suite:

- it can implement each DBMS-product separately from scratch,
- it can extend an existing system into one monolithic DBMS suitable for everything,
- it can try to come up with a set of reusable DBMS-components which are then assembled into different DBMSs, each fitting its respective application area.

The first possibility is obviously the most costly one and overly risky. The second possibility leads to a dinosaur, and such a successively extended system over time is becoming less stable and harder to maintain. Remains the third alternative, which is characterized by a component-oriented point of view. In this approach, an architectural model defines what the notion of “component” actually means, how components can be configured into a DBMS and which constraints have to be met upon configuration. The various components are either automatically generated out of a specification, or are implemented once, stored in libraries and successively reused for DBMS-constructions. In addition, a (software) *process model* defines how to proceed in order to configure a DBMS out of reusable components. Since this approach to a larger extent than others relies on modern software engineering techniques (software reuse, software architectures, and software process modeling, to name the three most relevant ones), we call this approach *methodical DBMS-construction*.

To a large extent, our motivation to consider the construction of DBMSs in a more scientific way stems from our past experience in building DBMSs (DAMOKLES [18], DAMASCUS [19], CoOMS/NO^2 [16, 27], and SAMOS [31]). In the first three, a from-scratch has been pursued, which meant implementing a lot of DBMS-parts that were not interesting for the respective aim of the project, but nevertheless have been necessary for a full-fledged DBMS. In the SAMOS project, a commercial DBMS has been used as a platform. Construction costs have been significantly lower in this layered approach, while this kind of construction also implied a lot of restrictions and required necessary work-arounds.

In special cases it will be feasible to extend an existing system or to implement a new DBMS from scratch, whereas in the general case a reuse-based approach is the only feasible one for implementing stable and maintainable DBMSs in a cost-effective way. For (academic) prototypes such a cost-effective approach would save resources and allow the implementors to focus on the interesting scientific issues. Additionally, as vendors are extending their product suite and going to offer multiple DBMSs, they will certainly also benefit from an approach that allows to reuse components accross the different DBMSs and to share common components.

In the KIDS-project, our aim is to develop such an approach to methodical DBMS-construction. KIDS contributes to DBMS-construction research in several ways, since it proposes an approach that is

- methodical,
- cost-effective, and
- open.

The KIDS approach is methodical since it applies recent software engineering results, such as software architectures, software process modeling, and—most important—software reuse. Due to its reliance on software reuse, it is cost-effective once a significant threshold of reusable artifacts exists. Finally, it addresses an initial set of DBMS-tasks (data definition and manipulation, transaction management, and integrity maintenance), but is open for further tasks.

1.1 KIDS in a Nutshell

The three major ingredients of the KIDS approach to DBMS-construction are its architecture model, the construction process, and its ability to cover multiple aspects. In this section, we give an overview of these basic concepts.

The objective of each construction is to obtain a DBMS with a certain functionality. The desired functionality is classified into *aspects*, i.e., each aspect defines a coherent subset of tasks and concepts of the DBMS under construction. Example aspects are data definition, storage, and manipulation (called the data model aspect) or concurrency control and recovery (the transaction management aspect).

The *architecture model* provides for constructs which are adequate to define the software architecture architecture of DBMSs. On a very abstract level, a DBMS-architecture is conceived as a collection of related brokers and services. A *broker* is a component being responsible for some of the DBMS's tasks. Each broker offers *services* to other brokers or to the DBMS-external world. Brokers can request services from other brokers; the rule-based implementation of brokers defines the reaction to such service requests. For each aspect, a set of services represents the functionality belonging to the aspect, and one or more brokers are designated as components implementing the aspect's functionality.

The *construction process* defines how to proceed in order to obtain a DBMS with the desired functionality. This process consists of several phases including requirements analysis and design, implementation, and integration of multiple DBMS-subsystems. Some phases of the process are common to all constructible DBMSs and independent of concrete aspects, e.g., requirements analysis and architecture design. For each aspect, however, a dedicated construction subprocess is defined and integrated into the enclosing DBMS-construction process. For each aspect, a specification language is used to define the DBMS-functionality belonging to this aspect. These specifications serve as input to aspect-specific implementation phases, which in turn use techniques such as generation of subsystems or configuration of subsystems out of reusable, already existing components.

1.2 Outline

The remainder of this paper is structured as follows. IN the following section, we describe our perspective of software engineering paradigms that are crucial for systematic DBMS-construction. The architecture model and the DBMS-construction process model are described in sections 3 and 4, respectively. Section 5 presents the KIDS approach to the construction of selected DBMS-parts. Section 6 reviews related work, and section 7 presents the current status of the project and conclude this paper in section 6.

2 The Software Engineering Perspective on DBMS-Construction

On a very conceptual level, DBMS-construction means to produce a specific piece of software system and thus can be regarded as an application of software engineering. Indeed, in KIDS we have experienced that methodical DBMS-construction can benefit from various software engineering paradigms. We therefore introduce our perception of the three most relevant ones in the subsequent sections.

2.1 Software Reuse

Software reuse means to build new software systems out of existing pieces of software (and possibly out of additional newly created ones). Software reuse is all but restricted to code fragments; virtually any result obtained in a previous system construction might be reused in later constructions (e.g., designs, specifications, test cases, etc.). We therefore commonly refer to reuse of *software artifacts*. Actually, designs are generally easier to reuse and have a higher payoff than reuse of code [5].

The appealing effect of software reuse is the decrease in construction costs: whenever a reusable artifact can be found, resources need not to be wasted for the repeated creation of it. Seen the other way, reuse is beneficial whenever the effort to define, select, customize, and integrate reusable artifact over time is less than always creating them from scratch. Since libraries or repositories of reusable artifacts have to be built before reuse can happen, reuse pays off only after several system construction (re)using these libraries.

Software reuse comes not for free, but instead has to be thoroughly fostered. Specifically, the following dimensions are crucial [39]:

- **Selection**: the retrieval of candidate artifacts for reuse in a concrete situation must be supported. Provided that libraries of reusable artifacts can be very large, effective reuse requires that finding the right artifact to reuse is faster than building it from scratch.
- **Specialization**: once an artifact has been chosen for reuse, it often needs to be specialized for the current context. This customization activity needs to be supported as well, and the modified artifact should be added to the library.

- **Integration**: typically, it will not be possible to select multiple reusable artifacts which then instantly form a new and coherent system. Instead, these selected artifacts need to be integrated and, again, support for this step is necessary.
- **Abstraction**: the key issue to reuse and the three aforementioned dimensions is abstraction. Reusable artifacts must be represented in an abstract way such that it can be easily conceived what their semantics are and whether they are appropriate for the current system under construction.

These dimensions can be addressed in various ways, e.g., according to the kind of reuse technique applied. Scavenging reusable code and designs [39] from existing systems appears as the natural form of reuse, though others can be imagined as well and pay off much more. For instance, generation or transformation of systems out of high-level specification can be considered as a kind of reuse as well; what is reused here is the overall design (which is fixed) as well as transformation rules. Ultimately, configuration of systems out of existing components is another form of reuse.

2.2 Software Architecture

A *software architecture* is a description of the system structure and decomposes the system into well-defined components such as subsystems and relationships between them. Software architectures are in turn formulated with the means of a *software architecture model*. Without a proper model of software architecture [47] a consistent DBMS will result only by chance. The most important ingredients of such an architectural model define the notion of component as the basic design units, relationship types between these components, and an architectural style (e.g., the style which considers DBMSs as being structured into five layers [37]). More precisely, an architecture model defines [47]

- constructs to define architectural *elements*, where three classes of elements are present in a software architecture:
  - processing elements,
  - data elements, and
  - connecting elements (the "glue"),
- an architectural form defining weighted properties and relationships:
  - weights indicate the relevance of a property or a relationship, or the necessity of selecting among alternatives,

    - properties are constraints defined on architectural elements, they serve to restrict the choice of elements, and
    - relationships are used to constrain the architectural elements in the way how they may interact and how they are organized with respect to each other.
- an architectural rationale motivating an architectural style, the choice of elements, and a form.

Software architectures themselves are related to reuse, since they actually form one specific reuse technique [39]. Software architectures represent significant design decisions that are reused as a whole. Moreover, the software architecture model should allow to plug further components (be they reused or newly created) into a system under construction.

2.3 Software Processes

The third aspect of software engineering relevant to DBMS-construction are *software processes*. A software process model defines explicitly how to proceed upon software system construction [53]:

> *A software process can be defined as a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products.*

Software engineering researchers and practitioners agree that the quality of resulting software product to a high degree depends on whether a proper software process model is defined and enacted during a construction [46].

Moreover, since the understanding of the software process itself is improved and requirements to it can change over time, *meta-processes* should be supported in addition. A meta software process is in turn a managerial process and defines how concrete process models can be adapted to changing requirements and contexts.

2.4 Software Engineering and DBMS-Construction

The three areas described above help to understand the open issues in DBMS-construction from a software engineering point of view. It should nevertheless be clear that DBMS-construction is not solely a plain application of these paradigms. Instead, paradigms evolved in software engineering still need to be adapted to the domain of interest, i.e., DBMS-construction. Not incidentally have domain-specific software architectures and application generators [42] been considered and found to lead to higher reusability. Furthermore, since adequate reuse techniques and useful architecture models depend on the domain they are applied to, proper *domain analysis* [48] is very important for a viable construction approach.

With respect to DBMS-construction, the overall challenge is to explicitly foster reuse for all phases of a DBMS-construction and all DBMS-parts. The different reuse techniques have different properties and are beneficial in specific situations, so that appropriate reuse techniques must be selected and adapted to the domain of DBMSs.

This challenge requires finding a domain-specific architecture model that is able to describe DBMS-architectures properly. This model should allow to reuse entire DBMS-architectures and to integrate components into such a concrete architecture.

Finally, a software process model is crucial for DBMS-construction, so that systematic, reuse-based development of DBMSs is supported from the very beginning phases and all relevant tasks of a DBMS are considered.

In the following sections, we will introduce a software architecture model, a DBMS-construction process model, and construction approaches to specific parts of a DBMS, all of which are smoothly integrated. Note that there is no separate section on reuse of DBMS-artifacts, since reuse pertains to virtually every activity in DBMS-construction.

3 The KIDS Architecture Model

In the previous section, we have stressed the importance of proper software architectures. This section introduces the KIDS architecture model which allows the definition of DBMS software architectures. In order to be a viable architecture model, it should provide for constructs to describe the architectural components of a DBMS and relationships between them. In addition, DBMS-architectures described in this model should

- be reusable themselves and
- be able to integrate the components that have been constructed in an aspect-specific manner.

Software architecture reflect major design decisions made for a software system. They should thus be reusable themselves, since designs yield a much higher reuse payoff then reusable code [5]. Furthermore, we have argued above that different reuse techniques will be advisable for the various parts of a DBMS, giving rise to aspect-specific construction approaches. The components constructed in such a way thus must be integrated into a coherent DBMS during a later phase of the construction process. To facilitate this step of integration, the architecture model has to provide for a degree of component autonomy as high as possible. If components are encapsulated as far as possible, components need to know only the minimal information about each other, and integration thus is eased. Second, architectural constraints should be supported that help to realize whether integration is correct and complete.

3.1 Architectural Style and General Decisions

Before presenting the architecture model we propose in KIDS, we will describe general decisions concerning the style of DBMS-architectures.

DBMS-architectures are often presented in layers [e.g., 1, 17, 45, 60]. The reference architecture model by Härder and Reuter proposes five layers [37]. This model is very well suited for understanding how a DBMS works, whereas it is actually hard to construct a DBMS as a stack of multiple layers and without messing up layers. Most systems thus end up with two or three layers, where the lower level in turn is a page-server (e.g., as in O_2 [17]) or a storage object manager (e.g., as in EXODUS [9], DASDBS [45]). As a first general decision, we thus decided to consider a DBMS as consisting of two layers, a low-level storage component and an upper layer implementing the data model, the transaction model, and further tasks. Thus, on the top-level view of a DBMS-architecture, the architectural style is a layered one.

Furthermore, an analysis of DBMS-architectures shows that the low-level storage components of a DBMS are quite similar to each other, whereas the real diversity lies in the upper levels (e.g., in those components that depend on the data model or the integrity model to a high degree). Therefore, if a generally applicable storage component could be found and used that satisfies the requirements of all constructible DBMSs equally well, the usage of this component would reduce the complexity of DBMS-construction significantly. The second design decision thus is to specify and implement such a general object storage server once and for all, to reuse it as an entirety within each DBMS construction, and to consider detailed construction of DBMS-components only for the upper layer.

Since the analysis of DBMS-architectures showed that the upper layer is in turn hard to decompose into layers, the question is which architectural style to choose for that layer. Since this layer is subject to reuse-based and aspect-specific construction, the architectural style should be characterized by a high degree of information hiding and ease of integration of separately constructed components to be plugged into this layer. The third design decision thus was to apply an event-based architectural style (in software architecture research also called "implicit invocation" [25]).

Although the intention of KIDS is the capability to consider any kind of DBMS-functionality, the final question is which kinds of DBMS-tasks or -components to consider right away (i.e., which aspects to elaborate on). While a large collection of aspects would make the project overly complex, too few aspects would not allow to properly evaluate our approach. We thus decided to consider three aspects in detail (the data model aspect, transaction management, and integrity maintenance). These initial aspects are reflected by a so-called *initial architecture* that populates the repository of reusable architectures right away and that can thus be reused in the very first DBMS-constructions.

3.2 The Broker/Services Model

In KIDS, we propose the broker/services model as a software architecture model for DBMSs. The prime concepts of this model are brokers, services, and responsibilities.

Services. A *service* represents a specific task to be provided by the DBMS[2] or a part thereof. Services are functional abstractions, they are comparable to functions of an abstract data type or methods defined for a class in object-oriented programming. A service has a *signature*, which in turn consists of a name and a list of typed formal parameters. Services can be requested by DBMS-components through the raising of *events*. In raising such an event, a requestor has to supply actual parameters according to the service definition. Each service is provided by at least one component in the system; these providers react to a service request with the service execution.

Each service defines (a set of) possible *replies*. Whenever a service can be provided successfully, one of the replies is raised by the service provider. That is, replies are raised like request events, and can have parameters as well. A reply can represent either success or failure of the service provision, which in turn is distinguished by reply names and parameters of replies. In addition, it exceptional situations it might be possible that the service cannot be provided at all in a meaningful way. Those situations are handled by *exceptions*, which are defined with the service and raised like other kinds of events.

2. Henceforth, we refer to a DBMS under construction simply as "DBMS".

Brokers. *Brokers* are the processing elements of a DBMS. Their major purpose is to enable the interaction between the DBMS-components, i.e. to request services and to satisfy requests for services. Brokers are comparable to objects in object-oriented programming, with the notable exception that they are *reactive*. Unlike a “passive” object, a broker is not invoked through receipt of a message or a function call. Instead, a broker reacts to specific events according to its definition. Events a broker reacts to are requests for services made by other brokers, or replies to a service the broker has previously requested.

Each broker has a state consisting of attributes. The state contains all the information a broker needs for providing its services. Particularly, a broker can contain sub-brokers, which in turn are responsible for a subset of the parent broker’s services, or which contribute to the provision of one or more services. A broker can also maintain references to programming language objects, which are needed to implement services. In contrast to brokers, those “passive” objects are invoked by usual message sending.

Whenever a request is made (i.e., an event is raised), the responsible broker “catches” the event and reacts by executing the service implementation. A service implementation consists of a set of event-condition-action rules (ECA-rules) [15, 21]. A ECA-rule in turn consists of an event (the triggering event, i.e., the request), a condition, and an action. Actions of ECA-rules are sequences of messages sent to the passive components of the broker, or generic operations such as replies. In the case that multiple ECA-rules are defined for the same event, their execution order can be constrained if necessary. The execution order is defined relatively through a partial ordering of ECA-rules.

Responsibilities. Finally, *responsibilities* are relationships between services and brokers; they define which broker provides for a specific service. See Figure 3-1 for the relationships betw-

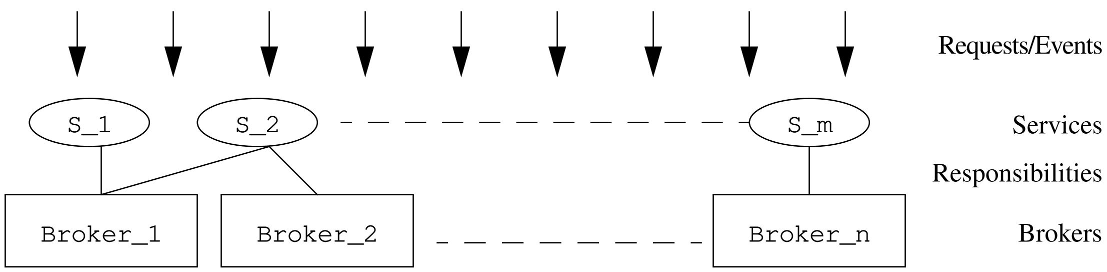

Figure 3-1 Relationships between Brokers, Services, and Requests/Events

```
BROKER broker_name                      SERVICE service_name
   [STATE {state_name: type}]             PARAMETERS {param_name: type}
RULES                                     REPLIES
   {RULE rule_name                        {reply_name, {param_name: type}}
   ON event                               EXCEPTIONS {exception_name}
   [IF condition]
   DO action
   [PRECEDENCE rule_name_list]}

              RESPONSIBILITY service_name: broker_name
```

Figure 3-2 Key Concepts of the Broker/Services Model

wen brokers, services, and responsibilities and the event-based architecture of KIDS. Figure 3-1 shows the key concepts of the model in pseudo syntax.

Brokers are grouped into *subsystems*. A subsystem is a collection of brokers which are responsible for a related set of tasks (e.g., object management, transaction management). In other words, a subsystem implements a certain aspect of DBMS-functionality in the aforementioned sense. Although the notion of subsystem is in principle redundant, it is nevertheless a convenient notation for talking about DBMS-structures and coherent sets of DBMS-components.

Example. As an example for how to informally use the introduced concepts, assume that a DBMS to be modeled should support management of objects in a multi-user and recoverable way. Let the DBMS thus consist of an object management subsystem and a transaction management subsystem. The former consists of a object management broker, while the latter consists of a transaction broker, a scheduler, a log management broker and a recovery broker. The transaction manager is responsible for starting and terminating transactions, it thus would be responsible for services `begin_transaction`, `commit`, and `abort`. Corresponding requests would be raised by DBMS-clients. The scheduler is responsible for serializing concurrent transactions, so it might be responsible for a service called `validate`, requests for which are raised by the object management broker.

DBMS-architectures. Based on the concepts introduced above, a DBMS-architecture is defined as follows.

Definition 3-1 (DBMS-Architecture)

A *DBMS-architecture* is a collection of brokers, services, and responsibilities.

Several kinds of generic architectural constraints have to hold for “nice” DBMS-architectures. *Completeness* holds if all required services are provided and implemented by some broker. For each service requested by some component, there must be a responsibility defined for that service and at least one broker. Each broker which is responsible for a specific service must define at least one rule for the implementation of that service. *Minimality* refers to the fact that each offered service is in fact requested by some other broker. Hence, minimality means that no “useless” services are offered by brokers. Finally, *syntactical correctness* holds if each requestor obeys the interface of the broker offering the service. Thus, the number of parameters and the type of these parameters of request and reply events must match the respective definitions of requests and replies in the corresponding service definition.

3.3 The Initial Architecture

The Broker/Services Model is designed to describe a broad range of DBMSs. While the architectures resulting over time can be reused in later constructions, a concrete architecture is desired that can be (re)used right away and that contains subsystems likely to be needed in most DBMSs. We call this concrete architecture the *initial architecture*, for obvious reasons. This architecture is the initial element of the architecture repository and can be reused even in the very first constructions, where no other architecture has been elaborated (and thus cannot be reused). The structural view of this DBMS-architecture is shown in Figure 3-3.

A DBMS-architecture consists of two layers. The bottom layer is formed by the object server component [43] which supports the storage and the retrieval of storage objects. It also provides for basic transaction management on storage objects. Such transactions for instance can form the lower layer of multi-level transactions [57] if object level locking and logging is desired (in this case, higher level transactions will be implemented on top of the object server). The object server provides a fixed interface, which hides implementation details and operating system characteristics from upper layers.

The object server is reused in its entirety, i.e., is itself not subject to construction for each DBMS. It thus belongs to the fixed part of a DBMS-architecture. This decision is justified since the object server implements functionality that is required by any DBMS. Second, its tasks can be implemented in a way that is adequate for any DBMS that uses it. Slightly changing requirements can be handled in that the object server is customized, while leaving its interface unchanged. Providing such a fixed, abstract object server interface means that

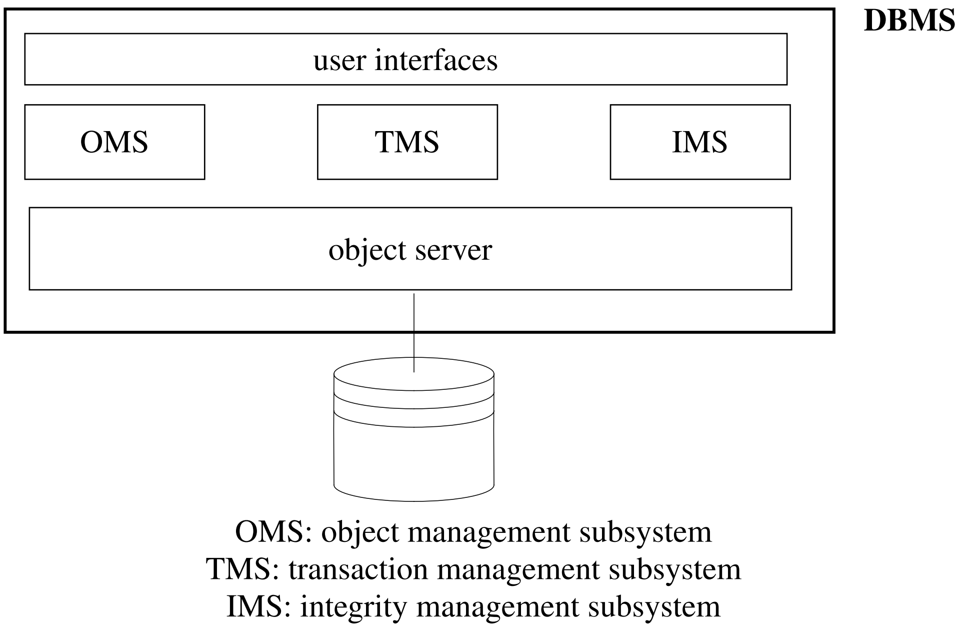

Figure 3-3 Initial Subsystem Architecture

constructions have to concern themselves only with the upper layer, and that the complexity of the construction process is significantly decreased.

On top of the object server is a second layer. Unlike the object server, this layer is variable to a large extent. It is decomposed into various subsystems. The initial decomposition proposes three major subsystems:

1. The *object management subsystem* (OMS) implements the mapping from data model objects into storage objects, retrieval of data model objects, and meta data management.
2. The *transaction management subsystem* (TMS) implements the concept of transaction, including concurrency control, recovery, and logging.
3. The *integrity management subsystem* (IMS) implements the (DBMS-specific) notion of semantic integrity. Its responsibility is to check whether database state transitions result in consistent states as well as to react appropriately in case of inconsistent transitions.

Due to the highly varying requirements to be met by distinct DBMSs, these subsystems cannot be implemented in a hard-wired way and be reused as a whole. Each of these subsystems will be designed and implemented using dedicated languages and reuse techniques (see section 5 below).

3.4 Example

In this section, we present an example of several brokers and a specific service (realization of transaction commit).

Assume that the previously mentioned brokers are defined for a DBMS: an object manager broker, a transaction broker, and an integrity broker. The example shows how IMS and TMS cooperate during the commit of a transaction. One prerequisite for the transaction to commit is that its modifications to the database yield a consistent state. To implement this constraint, the transaction broker is responsible for the service `commit`, and the integrity broker is responsible for the service `check`.

```
BROKER transaction_broker            // the transaction broker
STATE
    tm:   Transaction_Mgr; ...       // a passive object maintaining information about transactions
RULES
     RULE Commit-1
       ON    commit(tid: TAID)
       IF    not(tm->getState(tid) = "active")
       DO    raise(ILLEGAL_TA)
       PRECEDENCE Commit-2

    RULE Commit-2
       ON    commit(tid: TAID)
       DO    raise(check, tid, SYNC)        // invoke integrity check

    RULE Refused-Commit                     // reply
       ON    refuse(tid: TAID)
       DO    raise(abort, tid, ASYNC);      // abort due to inconsistent database state
             reply(refuse, commit)

    RULE Permit-Commit                      // second reply
       ON    agree(tid: TAID)
       DO    tm->commitTA(tid);             // commit transaction
             reply(ok, commit)

BROKER integrity_broker                     // the integrity broker
    STATE ...
RULES
    RULE CHECK_TA                           // check whether tid leaves DB in a consistent state
       ON    check(tid: TAID)
       IF    ...
       DO    reply(AGREE)                   // if yes, allow commit

SERVICE commit
    PARAMETERS   tid: TAID
    REPLIES      ok, refuse
    EXCEPTIONS   illegal_ta
```

```
SERVICE check
    PARAMETERS tid: TAID
    REPLIES          refuse, agree

RESPONSIBILITY commit: transaction_broker
RESPONSIBILITY check: integrity_broker
```

Example 1. Sample Brokers, Services, Responsibilities, and Rules

The transaction broker is responsible for the `commit` service. For each request, it checks whether the transaction to be committed is active at all (rule `Commit-1`). It then requests a check whether the transaction violates database integrity (rule `Commit-2`). IMS is responsible for this request. It replies either with the `refuse` or with the `agree` event. TMS in turn is responsible for these replies. In the former case, it aborts the transaction (rule `Refused-Commit`). In the latter case, the transaction is committed (rule `Permit-Commit`).

4 The KIDS Construction Process

The importance of a software process has been mentioned in section 2. In this section, we describe the software process model we have designed in KIDS for DBMS-construction. This process model defines the behavior of a construction support system, while the structure of this system is described by the meta architecture, to be presented next.

4.1 The KIDS Meta Architecture

A construction support system needs to be designed which manages collections of reusable artifacts and supports the construction process. The structure of this system is defined by the *meta architecture*.

A prerequisite for reuse is system support for storing and retrieving reusable artifacts. This task is accomplished by *repositories* [4] (*libraries*). Second, tools have to be provided that assist DBIs during a DBMS construction.

Repositories. Logically, each repository contains and manages artifacts of a specific type (e.g., architectures), though all repositories together can be stored physically in a single database. Repositories are defined for various kinds of artifacts:

- results of requirements analyses,
- DBMS-architectures,
- specifications (of certain DBMS-aspects),

- subsystem designs, and
- subsystem implementations.

The major purpose of the libraries is to store and support the selection of reusable artifacts. In order to reuse artifacts in constructions, the repositories must be populated. This can be done in two not exclusive ways. First, for each repository and kind of artifact, an *initial content* is provided. For instance, the initial architecture presented above is an element of the architecture repository. Furthermore, as more and more constructions are performed, these repositories are extended (section Figure 4-1). For each DBMS-construction, the artifacts produced during its course are added to the appropriate repository.

The artifacts are associated through relationships whenever they have been used or created within the same DBMS-construction. Requirements analysis results are linked to DBMS-architectures and specifications, specifications and architectures are linked to subsystem designs, and subsystem designs are in turn associated with subsystem implementations.

In addition, artifacts belonging to different aspects of a DBMS can be related to each other. Concretely, specifications of different aspects may need information about each other, and thus import/export-relationships between specifications are supported. For instance, a concrete specification of “consistent transition” belonging to the integrity aspect needs information about the types of transactions defined in the transaction model specification. Furthermore, designs/implementations of subsystems sometimes need information about each other and thus must also be able to reference each other.

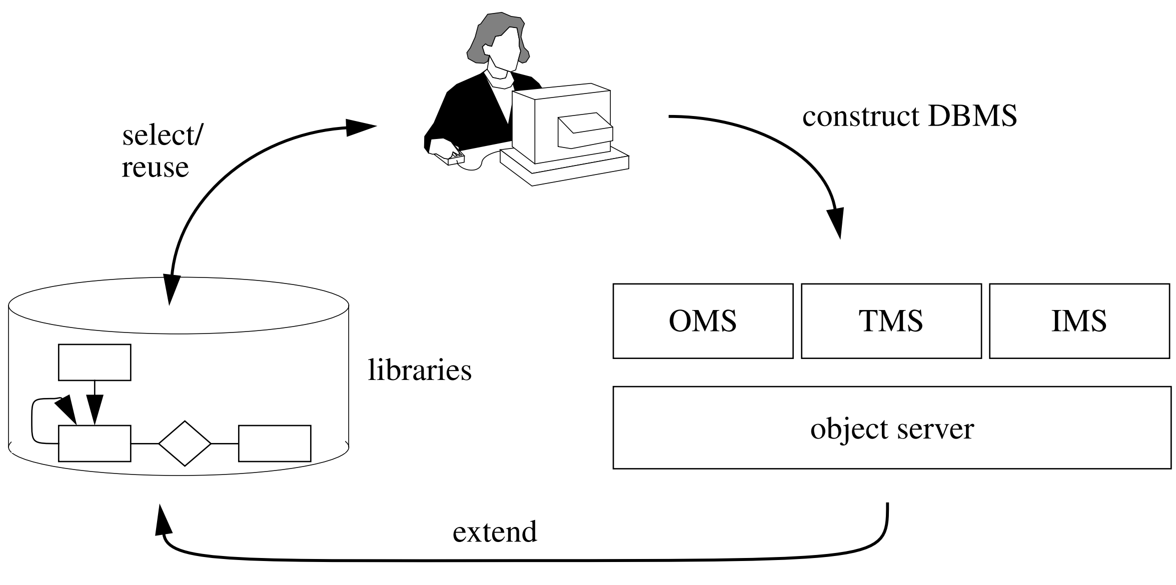

Figure 4-1 System Construction and Extension of Libraries

Tools. Tools are the second major ingredient of the meta-architecture. In principle, simple tools such as text-based editors or general programming language compilers would be sufficient, but would also leave more work to the DBI. Thus, in order to support the DBI during the construction process and foster reusability, the various repositories are used by *advanced tools* in addition to general-purpose tools such as text-editors or programming language compilers. These tools include:

- *editors* for specification of architectures or specific aspects,
- *compilers* for compiling specifications,
- *generators* constructing subsystems out of specifications, and
- *configuration tools* for searching reusable implementations and plugging them into subsystems.

These tools contribute to DBMS-construction in specific phases and possibly address different aspects of a DBMS, according to the construction process model. This model is described subsequently. Aspect-specific tools are presented in section 5.

4.2 Overview of the DBMS-construction Process

Figure 4-2 shows an overview of the activities of the DBMS-construction process and their interconnections. Boxes represent phases and the arrows represent the order to be preserved among the various steps. Shaded boxes express the fact that the corresponding activities are aspect-specific. Figure 4-3 shows the various repositories and the connections to the activities during which they are primarily used.

A prerequisite for the entire construction process is an understanding of the application domain characteristics in the DBI's mind. She is expected to know the application requirements for the DBMS to be constructed and have a rough understanding of the functionality of the desired DBMS. In other words, the construction process starts after a requirements analysis of the desired DBMS (for the specific application domain) has been performed.

In the first phase (DBMS analysis), a description of the DBMS functionality is given. The result of DBMS analysis is a list of features the DBMS is expected to provide.

In the second phase (architecture selection), a list of architecture skeletons is presented to the DBI. This list is annotated with the features it realizes, and the entire list is sorted such that architectures that support more features are ranked higher. The DBI then selects one of the architecture skeletons or decides to develop an own, new one. The result of this phase is an architecture skeleton, i.e., a (possibly still incomplete) collection of partially defined subsystems.

During the third phase (aspect analysis), the process branches such that each aspect is analyzed separately. The results of these (sub)phases are specifications of the various aspects. For each aspect, a specification language is provided. Each requirements analysis for an aspect is

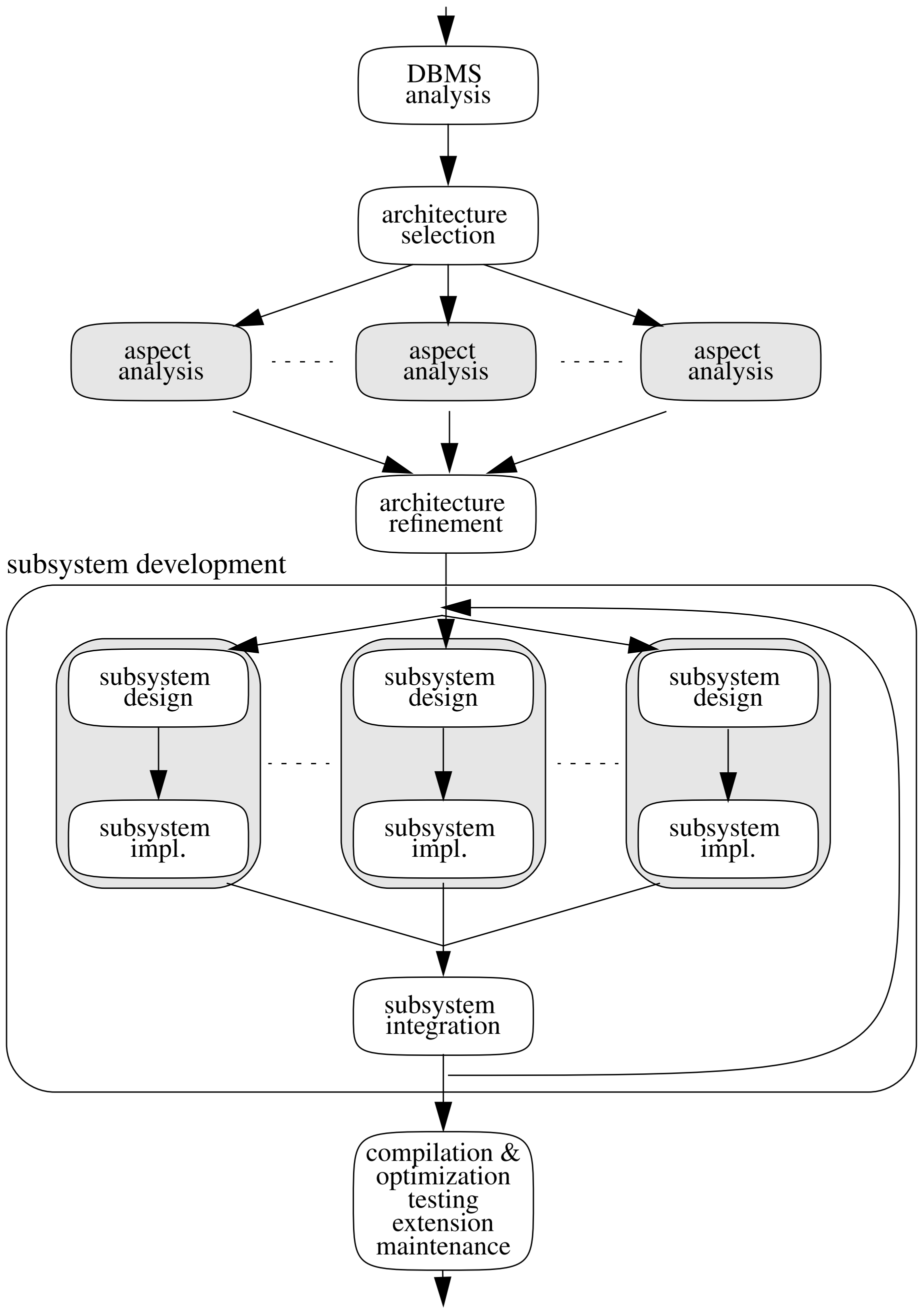

Figure 4-2 Phases of the DBMS-Construction Process

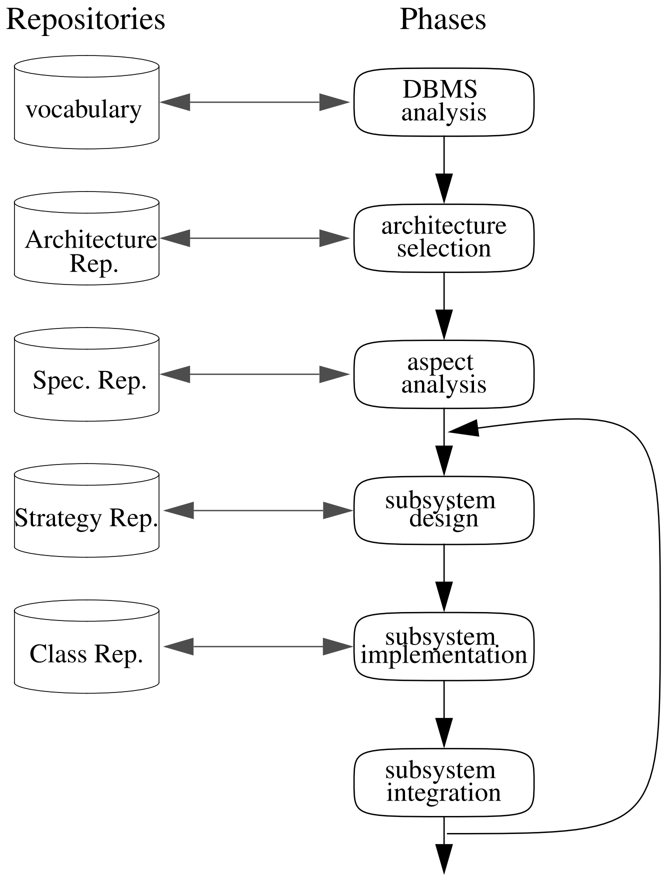

Figure 4-3 DBMS-Construction Process and Repositories

formulated in the corresponding specification language. Specifications do not have to start from scratch, rather existing ones may be reused in parts or as a whole.

Given the aspect specifications, the next phase consists of developing the corresponding subsystems. This phase in turn consists of three steps: subsystem design, subsystem implementation, and subsystem integration. Furthermore, the process branches not only in the sense of parallelism, but also in that two different approaches can be followed. Namely, subsystem implementation can be done by means of *generation* or *configuration*. If configuration is applied, reuse of classes that are stored in the class repository is supported. Mainly in the configuration approach (which is the one with more degrees of freedom), information on interfaces of other subsystems may be necessary. Hence, subsystem realizations must be coordinated at certain points in time. In this way, subsystems are integrated (i.e., linked together) to some extent. Since it might not always be possible to integrate the subsystems after the implementation of all the subsystems has been completed, it is possible to perform integration as far as necessary and then to continue with subsystem development. In other words, this phase is inherently iterative and incremental.

Generation, on the other hand, takes an aspect specification as input and automatically creates the subsystem implementation.

After subsystem development, four phases follow, some of which are optional. These phases are testing and evaluation, extension (implementation of user interfaces on top of the constructed DBMS), optimization, and maintenance. We will not investigate these phases in greater detail.

4.3 DBMS-Analysis

The first phase of the construction is *DBMS-analysis*. The goal of this phase is to capture general information about the desired functionality of the DBMS.

A DBMS-analysis is expressed in terms of *features* as proposed for classifying reusable software [48]. In our context, features are triples consisting of

- an aspect identifier,
- a subdomain identification, and
- a functionality identifier.

Aspect identifiers are the names of aspects (e.g., transaction management). The second component of such a triple identifies a logical subunit of the aspect, i.e., a subtask belonging to the aspect. For instance, within the aspect transaction management subdomains would be "structure of working units" or "transaction duration". The third component defines a concrete value for the subdomain, i.e. defines which concrete functionality is required from the DBMS. For instance, "transaction duration" can take "short" or "long-lived" as possible values.

All terms are collected in a *glossary*. A DBI can browse or query it in order to determine which features are known in the glossary and have been used in a previous analysis. Example 2 shows some sample features.

<transaction management, transaction duration, long-lived>	#1
<transaction management, transaction duration, short>	#2
<transaction management, structure of working units, nested transactions>	#3
<transaction management, structure of working units, flat transactions>	#4

Example 2. Some Features for DBMS-Analysis

4.4 Architecture Selection

The results of DBMS-analysis are used for two subsequent phases: *architecture selection* and *aspect specification*.

After analysis, the DBI selects an architecture skeleton for her current construction. Since the concrete functionality (for the desired aspects) is not yet fully known at this point in time, an architecture skeleton consists of a collection of partially defined brokers. However, such a skeleton already determines the relevant aspects as well as the set of brokers to be realized subsequently. Recall that the number of aspects may vary among different constructions.

The selection of reusable architecture skeletons is supported as follows. Each (previously designed) architecture is linked with its analysis. This relationship is used in order to determine architecture candidates for the current construction. Previous architectures are ordered according to the number of features (of the new analysis) they realize. The DBI selects one architecture out of this list. Depending on whether the reused architecture skeleton covers all the features in the current analysis, the DBI may have to extend the architecture skeleton for the current construction.

As an example for a reusable architecture skeleton, consider the initial architecture in Figure 3-3.

4.5 Aspect Specification

Analysis information is also used to retrieve reusable artifacts in the next phase: the *aspect specification phase*. Although the vocabulary is controlled, the analysis in the first phase is still on a quite informal and general level. During aspect specification, however, the functionality of each aspect will be formally specified.

Obviously, the various aspects are characterized by quite distinct properties. Therefore, we suggest to use aspect specific languages for specification, instead of using a general purpose specification language, such as FOOPS [32] or OS [8]. In this way, specification languages are tailored to their corresponding aspect, and aspects can be specified much more compact and naturally. We will describe concrete specification languages in more detail in section 5.

Specifications are in turn reusable, in parts or as a whole. For that matter, aspect specifications are also linked to their DBMS-analysis. Similar to the architecture selection, reuse candidates are determined and selected.

4.6 Architecture Refinement

After aspect analysis, the next phase is *architecture refinement*. Assume that some aspects will later on be implemented by generators based on the aspect specifications. Typically, the implementation will rely on the functionality provided by other subsystems and will also provide for functionality used by other subsystems. The underlying observation is that subsystem implementations cannot be arbitrarily combined. As a result, the implementations will have to integrate well, although for generated parts, there is in general little degree of freedom to afterwards modify the implementation. Therefore, during subsystem refinement, generators enhance the architecture such that the modified architecture reflects the conditions imposed by the generated subsystems on other components.

Hence, the purpose of this phase is to enhance the definition of brokers as far as possible in order to prevent dead-ended designs. New services or sub-brokers can be defined in this phase, responsibilities can be established between brokers, and activities for service implementations containing requests can be defined. The information required for these modifications is derived from the aspect specifications and is provided by the generators.

4.7 Subsystem Development

After architecture refinement, the subsystems (each implementing an aspect) are constructed. This phase is structured into three subphases. First, a subsystem design may still have to be performed. Subsystems are then implemented, and all subsystems will be integrated in the last subphase. Depending on the aspect under consideration, two different techniques are applicable: generation or configuration. The former takes an aspect specification as input and automatically creates executable code that implements the specification. Configuration also takes a specification as input, but implements this specification by composing reusable artifacts into a subsystem. In section 5 below, we show that this distinction is useful and necessary, and also describe which reuse technique is applied for which aspect.

Subsequently, we describe each of the aforementioned subphases of subsystem realization.

4.7.1 Subsystem Design

Depending on the implementation technique applied for a specific subsystem, *design* is the first subphase. Subsystem design defines the internal structure of brokers. Hence, if generation is applied, this structure is determined automatically by the generator. In this case, therefore, subsystem design is not necessary. If configuration is applied for the subsystem, the subsystem still has to be designed before implementation.

In general, configuration is used whenever a large variety of implementation alternatives exists for the subsystem, and the "best" choice cannot be determined automatically. In such cases it is an apparent problem of design to choose among the spectrum of implementation possibilities. As an example, take transaction management in general and concurrency control in particular, where a large variety of different implementations exist, and the appropriate one depends on the intended application domain.

Obviously, decision support during subsystem design is of great help for a DBI. In the KIDS construction process, such decisions are supported through the notions of *strategy* and *technique* [30]. During subsystem design, the DBI

- receives hints which design decisions are to be made at all,
- is provided with a classification of alternatives for each necessary design decision,
- is assisted in preventing inconsistent designs and incompatible decisions.

For that matter, each aspect is structured into several *dimensions*. A dimension represents a design decision that must be made during subsystem design. Within each dimension, *strategies* are the alternative design decisions that can be made. In other words, for each dimension of an aspect, exactly one strategy has to be selected. Strategies are defined formally as first order formulas over brokers and services and their related concepts. Additionally, *incompatibility* between strategies is explicitly expressed. The meaning of this relationship between two predicates is that the two related strategies may not be chosen together within the same construction, since this combination would lead to incorrect system behavior.

Strategies are abstract representations useful for representing design decisions. They are *implemented* by techniques.

As an example, consider the concurrency control domain. One dimension of this aspect is the *validation time*, i.e., the relative point in time when a database access on behalf of a transaction is determined to be legitimate or not. Strategies within this dimension are *preclaiming* (validation at transaction begin), *pessimistic* (validation directly before database access), and *optimistic* (validation at transaction commit). Each of these strategies has impacts on the service interfaces, order of requests, and so forth. These restrictions are expressed through the formal definition of the strategies. One strategy for the object management aspect is *update in* *place*, and the fact that update in place is not allowed for optimistic concurrency control is expressed through the incompatibility between the two strategies.

4.7.2 Subsystem Implementation

Subsystem implementation follows the subsystem design phase. In this phase, concrete classes (in the sense of object-oriented programming) are selected that implement the behavior of the subsystem. This may be done automatically, as in the case of generators, or semiautomatically, as in the case of configuration.

In configuration, selection of reusable artifacts is supported through the notion of *technique*. A technique is the implementation of a combination of strategies (of different dimensions). This relationship is expressed explicitly, so that once strategies have been selected, the a list of techniques for implementation can be proposed to the DBI. Additionally, each technique specifies *conformances*, which determine how the technique implements its strategies. Conformances specify which method implements which service.

Techniques are object-oriented classes which can still be parameterized with other classes. The basic components are *primitives*. A primitive is an object-oriented class as well. In other words, a technique is a blueprint for implementation in terms of other techniques and primitives. Both primitives and techniques are formally specified, using Meyer's *design by contract* [41]. Each class specifies a *class invariant*, i.e., a formula that has to hold for each instance of this class. Additionally, pre- and postconditions are specified for methods. Preconditions specify requirements that have to hold in order to correctly execute the method. Postconditions specify assertions that hold after the method has been executed. Invariants as well as pre- and postconditions are necessary in order to formally capture the semantics of classes, which is important during modification of the technique and the repository of primitives (i.e., the addition of new or the modification of existing primitives and techniques).

During subsystem implementation, techniques are proposed to the DBI based on the strategy selection. The DBI selects a collection of techniques and includes them in the current subsystem design, until the subsystem is fully implemented.

4.7.3 Subsystem Integration

Different subsystem implementations for distinct aspects still have to be integrated, because subsystems can be implemented independently from each other and possibly in parallel. Therefore, the *subsystem integration subphase* is necessary to complete the subsystem development phase. The objective of this phase is to ensure architectural constraints, such as completeness, minimality, and (syntactical) correctness (see section 3.2).

Finally, note that we have described subsystem development as a linear sequence of subphases. In reality, this phase will be cyclic. It will be reasonable in some cases to perform a partial design, to select techniques, then to integrate the existing part with other already existing parts of the DBMS and then to proceed with subsystem design.

After the final integration step, the implementation of the DBMS core is completed. The following phases will perform activities such as testing, implementation of user interfaces, and maintenance. Although these phases are very important, they are not explicitly supported within the current construction process, and therefore are not discussed here.

4.8 The KIDS Meta Process

The KIDS meta process defines how new aspects can be introduced to the construction method in a systematic way. For instance, in the current state of KIDS, authorization/access control would be such a new aspect.

The KIDS meta process defines the following steps:

1. Perform a domain analysis of the new domain of interest.
2. Identify fixed parts of the aspect and make assumptions explicit.
3. Develop an aspect specification language that is declarative and comprehensive for DBMS constructors (i.e., uses well-known terms from database technology).
4. Modularize the specification language as far as possible.
5. Determine knowledge required from other aspect specifications (i.e., imported information).
6. Decide for a suited implementation approach: generation or configuration.

Results of executing these steps for the data model, transaction management, and integrity maintenance are presented in section 5.

The last step is governed by a set of heuristics. In general, we conceive subsystem implementations as mappings from higher level constructs and tasks (e.g., data model objects, transactions) to more primitive ones as, e.g., provided by the object server. Depending on what is known beforehand about these mappings, configuration or generation should be chosen. Namely,

- if the mapping is uniquely defined as a function of the upper level specification, generation is feasible,
- if multiple such mappings exist but the “best” one can be determined algorithmically, generation is also feasible,
- if a large number of mappings (implementations) exist for an aspect and none of them is known to be best, configuration is better suited,
- if the domain knowledge is still rapidly evolving, then configuration should also be preferred.

It is also possible that for a concrete aspect a mixed approach is followed. In this case, the aspect is decomposed in subdomains, some of which are implemented using a generative approach, while others follow the configuration approach.

5 Construction of Selected DBMS Aspects

5.1 The Data Model Aspect

The aspect of data models (or object models) covers the mapping of data model objects into storage objects, retrieval of data model objects, and meta data management. Major tasks within this aspect includes the specification of the data model and the construction of an adequate object management subsystem. For specification, we designed a data model specification language [50] called DMSL permitting the DBI to define the algebraic syntax and semantic of the desired data model. The construction of OMS follows a generative approach.

A data model specification syntactically comprises a set of concrete types and generic functionality (operations) that is applicable to instances of such types. Different sorts of concrete types are distinguished according whether they show object-based, value-based, or association-based characteristics. Unlike value-based types, the instances of an object-based type expose identity. A type is called association-based if its instances model relationships between entities.

In the relational data model, concrete types might be `RelationSid` referring to relational schemas and `RelationIid` to relational instances that can be constructed by applying an instantiating operation to instances of `RelationSid`. Generic functionality applicable to instances of `RelationIid` might be the usual relational operators such as join, union, projection, etc.

Related types and their specific generic functionality are grouped into subspecifications. Reuse is facilitated by reusing non-parameterized subspecifications or by instantiating parameterized subspecifications involving the sophisticated DMSL parameterization mechanism. Finally, a top-level specification enumerates the subspecification relevant for the desired data model.

After having defined the interface of the data model, the DBI is required to specify the semantics of the data model by providing an implementation of each operation included in the generic functionality of the data model. For that purpose, the DBI may refer to operations in existing specifications (maybe lower level specifications, finally using the operations of the object server shown in Figure 3-3). The flow of control is defined by a typed functional sublanguage of DMSL.

Once the specification of the data model is completed, the design and implementation of the object management subsystem follows. In the current KIDS prototype, a generative approach to subsystem design and implementation was adopted with the goal to let the DBI focus on the specification phase rather than on design and implementation.

Specifications in terms of DMSL serve as input of the data model specification tool responsible for checking data model specifications for completeness and correctness. Subsequently, the DMS generator transforms checked data model specifications into operational data management subsystems. The portions of the DBMS construction system that are of interest in this context are illustrated in Figure 5-1.

We conclude this section by illustrating the top-level DMSL specification of a structurally object oriented data model [20] called NO^2 [27]. The specification indicates that the NO^2 type system includes some meta value set specification instantiations that are not parameterized (`Integer`, `Real`, `LongField`, `Char`, `String`) and some parameterized ones (`Set`, `List`, `Tuple`, `Array`). With respect to the instantiation of the specification `List`, list values and list value sets are specified as being constructed from any kind of elements that may result from the application of the concepts defined in the meta value set specifications as instantiated in Example 3, #2-#5. The keyword `VALUE SETS` as actual instantiation parameter of the specification `List` tells us that all specifications mentioned in the `VALUE SETS` section of data model specification NO^2 shall be included as actual parameters. In this (recursive) manner, a type system is defined that permits the schema designer to orthogonally construct values later on.

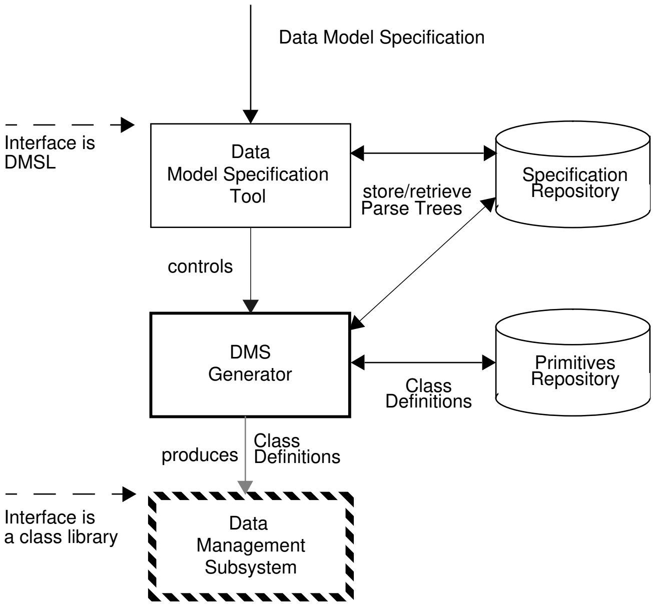

Figure 5-1 DMSL related components of the DBMS construction system

Moreover, the data model specification NO^2 states that one kind of object types is available by instantiating the meta object type specification called `ObjectType`. The actual parameter `VALUE SETS` suggests that states of objects can be modeled by any kind of value which can emerge by applying the concepts provided in the `VALUE SETS` section of the data model specification NO^2.

Furthermore, the DBI instantiates a previously specified meta association specification called `ER` with two actual parameters (Example 3, #9). While the first parameter denotes the concepts that can be connected by this kind of meta concept, the latter defines the possible structures for modeling the states of individual relationship instances. Specification `ER` defines a general relationship concept with abilities as described in most textbooks, e.g. [22]. Note that

```
DATA MODEL NO²                                                                    #1
    VALUE SETS                                                                    #2
        Integer, Real, LongField, Char, String,                                   #3
        Set[(ObjectType, VALUE SETS)], List[(ObjectType, VALUE SETS)],            #4
        Tuple[(ObjectType, VALUE SETS)], Array[(ObjectType, VALUE SETS)]          #5
    OBJECT TYPES                                                                  #6
        ObjectType[(VALUE SETS)]                                                  #7
    ASSOCIATION TYPES                                                             #8
        ER[(OBJECT TYPES), (VALUE SETS)], SUB                                     #9
END                                                                               #10
```

Example 3.A Top-level Data Model Specification general relationships are in fact not supported by NO^2, but subobjects are. Specification `SUB` describes the subobject concept. `SUB` is directly based on `ER`. For full details, refer to [50].

5.2 The Transaction Model Aspect

The transaction model aspect determines the units of work submitted to the DBMS, and it implements concurrency control and recovery [3, 35].

The specification of transaction models in KIDS is supported by a language called SPECTRA [29], which extends ACTA [13, 14] and—unlike ACTA—defines a formal syntax. The major concepts of SPECTRA are transaction types and dependencies. A *transaction type* defines the structure and behavior of its instances, mainly in terms of events (requests), states, and transitions. *Events* are the requests a transaction can legitimately issue, or that can be issued for that transaction. They are grouped into initiation (e.g., `begin_transaction`), positive (e.g., `commit`) and negative (e.g., `abort`) termination, and further (e.g., database operations) events. States model the different stages in the lifetime of a transaction. *Transitions* are quintuples `(s,e,s',pre,post)`, with the following meaning: if a transaction is in state `s`, `e` is requested and the precondition `pre` holds for it, then it transits in state `s'` and the postcondition `post` must hold. Preconditions are logical predicates that are required to hold in order to allow the transition, while postconditions can establish new facts for the transaction. Most prominently, dependencies can be established in postconditions.

The notion of *dependency* is borrowed from ACTA. Dependencies express relationships between transactions; they are defined as logical formulas that must hold for the related transactions. The *abort dependency* is an example: if a transaction `t` is abort-dependent on another one (`t'`), then `t` must abort whenever `t'` aborts.

For full details of SPECTRA, see [29]. Example 4 shows parts of the specification of Sagas [24].

```
specification Sagas
dependencies
  // AD, WD, WCD, ...: some dependencies
variables
      saga: Saga;
definitions
       conflicts(read,  write), (write, read), (write, write)
       predicates
          C(t1:Transaction, t2:Transaction) := ! t1 = t2 &
              exists ob:Object (exists p:Operation (exists q:Operation
                          (conflict(p(t1, ob), q(t2, ob) ) & p(t1, ob) -> q(t2, ob))))
```

```
        //defines the notion of conflict

// The Saga transaction type
transaction type Saga
...
. // end Saga

// The Component transaction type
transaction type Component_TA
    t: Component_TA
    properties
        duration: short
        view-set := current
        conflict-set := { tx:Component_TA ! tx = t & active(tx)} + {tc: Compensating_TA active(tc)}
    events
        {begin(saga, t)}, {commit(t)}, {abort(t)}, {op}
    states
        {active, committed, aborted, terminated=(aborted, committed)}
    transitions
        ( , begin(saga, components[i]), active)
            precondition terminated(components[i-1])
            postcondition
            t = components[i] & AD(saga, t) & WD(t, saga) & BCD(compensations[i], t);
        (active, op, active);
        (active, commit, committed)
            ...
        (active, abort, aborted)
. // end Component_TA

// The Compensating transaction type
transaction type Compensating_TA
...
. // end Compensating_TA
end Sagas
```

Example 4. SPECTRA Specification of Sagas

A transaction model specification determines a great part of the transaction management subsystem interface, i.e., its services. For many services, however, it will still be open how they are implemented. This is determined during subsystem design and implementation.

The TMS-design is refined during the subsystem design phase. The starting point are the transaction types and services defined for them. In general, each event defined for a transaction type results in a service definition of TMS. Further services might be derived as an outcome of the specification, e.g., services for `register` (recording a database access on behalf of a transaction) and `validate` (checking whether the database access is legitimate with respect to the execution of concurrent transactions).

Some subtasks of transaction management can be implemented in quite different ways. This is typically the case for concurrency control and—to a more limited extent—for logging and recovery. Given that large variety of alternative implementations, KIDS makes use of the concepts of strategy and technique described above. First, the relevant dimensions are determined, and one strategy is selected for each of them. Once the selection of strategies is complete and fixed, subsystem design is completed as well.

Currently, seven dimensions have been identified for the concurrency control aspect:

- *validation time*: when is the database access validated (e.g., directly before the access)?
- *validation reaction*: what is the reaction in case validation fails (e.g., abort the transaction)?
- *validation basis*: which information is taken into account for validation (e.g., a generic conflict relation, or the transaction structure)?
- *granularity*: what is the unit of validation (a single object, or multiple objects as in multi-granularity locking)?
- *data dependency*: does validation depend on the value of accessed data items or parameters (e.g., as in Escrow-locking [44])?
- *type dependency*: does validation take the semantics of data types into account (e.g., as in semantic concurrency control)?
- *versioning*: does validation apply a single- or a multi-version approach?

Consider the first dimension (validation time). Possible strategies are `preclaiming` (validate at transaction begin), `pessimistic` (directly before the database access), and `optimistic` (before commit). The selection of strategies (for one subsystem as well as across subsystems) is constrained through *compatibilities*. Two strategies of different dimensions may not be chosen together if they are related through the predicate `incompatible`. For instance, the `optimistic` strategy is incompatible with `update-in-place` in the object management subsystem. Incompatibilities must be specified when strategy definitions are added to the strategy repository.

Based on the selection of strategies, the DBI then selects one or more techniques that implement the combination, if there are any in the repository. Such techniques (e.g., the strict two-phase locking protocol for concurrency control) are composed out of classes. They are determined during subsystem implementation and are added to the subsystem code. The knowledge which strategies are implemented by which techniques is specified upon the definition of new strategies/techniques; it is used during subsystem design in order to assist the DBI in finding feasible techniques for the chosen strategies.

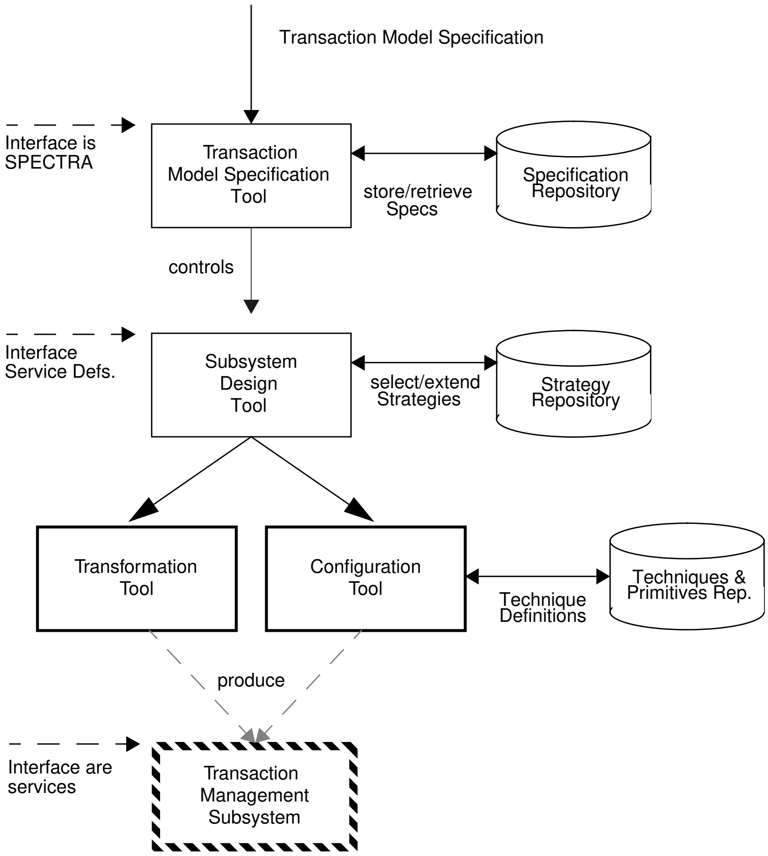

Figure 5-2 SPECTRA Related Components of the DBMS Construction System

The reuse technique applied for concurrency control is configuration. For other tasks of transaction management, transformation is feasible. Recall the notion of dependency introduced above, which is a case where this approach is pursued. The transaction management broker is responsible for establishing dependencies as soon as they are formed (the point in time when they are formed is given in the postcondition of transitions). Enforcing dependencies is implemented by the transaction manager as well. For implementation, the definition of a dependency is rewritten in terms of event-condition-action rules. For example, let `dep(`t_1`,`t_2`)` be such a dependency, with the definition that when t_1 issues the request `e` then t_2 has to react with the service `s`. The corresponding rule then is to enforce the dependency upon `e`'s occurrence and to request `s` for t_2 in the action of the rule. Alternatively, the definition of a dependency can mean that t_1 can only request `e` when `s` has happened for t_2 before. This condition is checked for t_2, if it does not hold, t_1's request is rejected. For full details of these transformations, see [26]. Figure 5-2 shows the relevant parts, activities, and repositories of the construction support system with respect to transaction management.

5.3 The Integrity Maintenance Aspect

An *integrity model* is a collection of explicit integrity concepts allowing to define and enforce explicit constraints that scope more than one simple data model operation; thus taking the semantics of database state transitions (temporary inconsistencies, constraint simplification) into account.

An integrity model is characterized by the following facets:

- *Transitions*: they represent the sequences of operations that will be considered as transitions by the desired integrity subsystem.
- *Conditions*: they define inherent and implicit constraints in terms of DMSL. For explicit constraints, however, the DBI must define the necessary logical language and how sentences in this language can be (efficiently) evaluated with respect to transitions.
- *Reactions*: The DBI ensures the non-violation of inherent and implicit integrity constraints by correctly imlementing the data model operations in terms of DMSL. Additionally, the DBI has to specify the reactions to execute in case of explicit constraint violations.

For the specification of these facets we provide sublanguages of our general integrity model specification language called IMSL.

An example integrity model specification is shown in Example 5. The intention of the `TERMS`-clause is to characterize the set of permitted terms that will be supported by the desired logical language. On the one hand, reference is made to the corresponding data model specification (`RelDaMo`), that is, all instances of the concrete types introduced by the referenced data model serve as an initial set of terms. On the other hand, more complex terms can be defined

```
INTEGRITY MODEL RelInMo              /* its finally a relational integrity model... */
TERMS
    CONCRETE TYPES FROM RelDaMo /* ... referencing the data model specification RelDaMO */
    FUNCTION SYMBOLS
            >, ∧, ¬,          /* basic functions */
            ∀RelationI,       /* universal quantification of onerelation instance */
            ∀∀RelationI,      /* universal quantification of two relation instances */
            ∃RelationI,       /* existential quantification of one relation instance */
            ∃∃RelationI,      /* existential quantification of two relation instances */
            ∀∃RelationI,      /* univ./exist. quantification of two relation instances */
            count             /* a general aggregation function */
REACTIONS UserDefined         /* reaction type that permits user-defined reactions. */
TRANSITIONS FlatTransactions  /* reference to transition specification.*/
END
```

Example 5. Application of the Top-level Language Concept by applying function symbols to existing terms. A `function-symbol-name` refers to a separate subspecification where its interface and its semantics (how the application of a function symbol to existing terms can be evaluated) are specified (will be explained below).

Terms can be constructed compositionally by applying function symbols to existing terms. The set of possible conditions is simply the largest subset of terms such that each term evaluates to boolean value (more technically, a `Boolean.BooleanVid`). Note, however, that the definition of terms occurs primarily at DBMS execution time; i.e. at constraint definition time.

The `REACTIONS`-clause defines the permitted reactions that can be specified by the schema designer later on for recovering from inconsistent database states. A `reaction-type-name` (Example 5, #12) thereby refers to a separate subspecification. For details on the IMSL-concepts for specifying such reaction specifications, see [50].

Finally, the `TRANSITIONS`-clause defines the set of transitions that are relevant for the desired integrity model by referring to the corresponding subspecifiation.

Thus, the integrity model `RelInMo` comprises assertional and referential conditions taking the semantics of state transitions into account. In addition, a counting aggregation function is included as well. State transitions shall correspond to flat transactions which are assumed to be specified in a transaction model specification called `FlatTransactions`. In case of constraint violations detected at the end of transitions, the desired integrity model shall permit arbitrary DBA-defined reactions.

Let us now turn to a subspecification defining a function symbol. In Example 6, a function symbol with universal-quantifying semantics is specified. Using this function, it can be specified that a certain predicate `p` (Example 6, #5) must hold for all tuples of a certain relation identified by `rid` (Example 6, #4).

In the rather radically functional view proposed here, quantifiers are modeled by function symbols similarly to basic predicates. The evaluation of $\forall_{\texttt{RelationI}}$ denotes whether a unary predicate `p` holds for all tuples of a particular relation instance `rid`. Both, `rid` and `p` are bound variables. While the former accepts instances of the concrete type `Relation.RelationI` as value, the latter accepts operations representing mappings from `Tuple.TupleVid` to `Boolean.BooleanVid`.

The various clauses of such a specification have the following meaning:

- The FUNCTION clause defines the name of the function.
- The RANGE clause defines the return type of the function.

- BOUND VARIABLES specifies the names and types of input parameters.
- The COMPLETE EVALUATION clause specifies an evaluation method for the entire relation instance in one step (i.e., no incremental evaluation). `Relation.forAllTp` is a data model-specific operation; it returns `Boolean.TRUE()` if the given predicate holds for all tuples of the relation instance.
- The INCREMENTAL EVALUATION section defines how the predicate can be evaluated incrementally, i.e., only for those tuples that have been added or modified during a transition. Several clauses are necessary to specify incremental evaluation:
  - The net effect of a transition is represented by a delta relation, which is computed using ECA-rules (the rule in #18-#20 inserts a new tuple into the delta, and the rule in #22-#24 removes a deleted tuple from the delta).
  - The INITIAL clause specifies the state of the delta at the beginning of a transition, and defines the constraint under consideration to hold at the beginning of a transition.
  - The FINAL clause defines how the validity of the function is evaluated at the end of a transition. In the example, it is specified that the same evaluation function as for complete evaluation is used, however, it is only applied to the delta relation.

```
FUNCTION SYMBOL ∀RelationI              /* referenced in (#6) */                          #1
RANGE Boolean.BooleanVid                                                                   #2
BOUND VARIABLES                                                                            #3
    rid: Relation.RelationIid              /* id of a relation instance */                 #4
    p: (Tuple.TupleVid): Boolean.BooleanVid    /* a mapping from tuples to boolean */      #5
AUX VARIABLES                                                                              #6
    δrid: Relation.RelationIid             /* delta relation recording the deletions on rid */   #7
    b: Boolean.BooleanVid                  /* accepts results of relation manipulations */ #8
COMPLETE EVALUATION                                                                        #9
    THIS:= Relation.forAllTp(rid, p)                                                       #10
INCREMENTAL EVALUATION                                                                     #11
INITIAL                                                                                    #12
    THIS:= Boolean.TRUE()                                                                  #13
    δrid:= Relation.createRelationI(String.aUniqueStringV(), Relation.getRelationSbyInstance(rid))   #14
FINAL                                                                                      #15
    THIS:= Relation.forAllTp(δrid, p)                                                      #16
                                                                                           #17
EVENTRelation.insert(ri: Relation.RelationIid; tv: Tuple.TupleVid; RES: Boolean.BooleanVid)    #18
WHEREBoolean.and(RES, EAV.identityp(rid, ri))                                              #19
ACTION b:= Relation.insert(δrid, tv)                                                       #20
                                                                                           #21
EVENT Relation.delete(ri: Relation.RelationIid; tv: Tuple.TupleVid; RES: Boolean.BooleanVid)   #22
WHERE Boolean.and(RES, EAV.identityp(rid, ri))                                             #23
ACTION b:= Relation.delete(δrid, tv)                                                       #24
END                                                                                        #25
```

Example 6. Universal Quantification over Relation Tuples

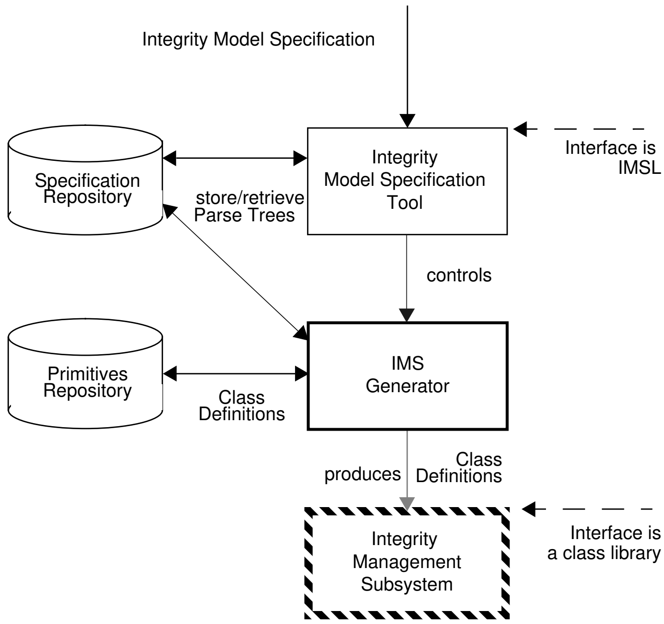

Figure 5-3 IMSL related components of the DBMS construction system

Note that incremental evaluation using net effects of transitions allows inconsistencies during a transition. Furthermore, incremental evaluation is likely to operate on a relation instance `δrid` of smaller cardinality than `ri` used for complete evaluation.

Once the specification of the integrity aspect is completed, subsystem design and implementation starts in a fashion analogous to the construction of the object management subsystem: Specifications in terms of IMSL serve as input for the integrity model specification tool responsible for checking their completeness and correctness. Consequently, the IMS generator transforms checked integrity model specifications into operational integrity management subsystems. The portions of the DBMS construction system which are actually relevant are illustrated in Figure 5-3.

6 Related Work

Database research has proposed the concept of “extensible database system” [11] for the development of DBMSs. We find this term misleading, since it is not always a full-fledged DBMS which is extended, and sometimes the term “extension” is not appropriate. We therefore prefer to talk about “construction approaches”.

Construction approaches can be classified in two different ways:

- according to the style of construction and the construction phases that are effectively supported,
- according to the aspects they address.

The following styles can be distinguished:

- kernel approaches,
- customizable DBMSs,
- specification frameworks,
- DBMS-generators,
- configurable DBMSs.

A DBMS-kernel offers a more or less fixed functionality, which implements the lower layers of a DBMS. Typically, the upper layers must be coded manually, and there is little support for the realization of these layers. Example systems include EOS [6], BeSS [7], the storage manager of EXODUS [10], WISS [12], ObServer [38], KIOSK [43], the DASDBS kernel [45], KALA [51], and TEXAS [52]. Notwithstanding that in the pure kernel approach efforteous manual coding is necessary, kernels have written the true success stories so far: the IRIS object-oriented DBMS has been implemented on top of the storage manager of a relational DBMS [60], O_2 uses a modified version of WISS, ODE is implemented on top of EOS, and Open OODB uses the storage manager of EXODUS.

Customizable DBMSs can be modified/extended at specific, well-defined places in the system (e.g., attachments in STARBURST). In this approach, the realm of constructible DBMSs is rather restricted (e.g., due to a fixed data model). A typical representative of this kind of system is STARBURST [40], which has been used, e.g., for the implementation of an active DBMS [59].

Specification approaches allow to model the semantics of some aspect, but do not necessarily address the implementation of these specifications. Representatives are SOS [36] for the data model aspect and ACTA [13] for the transaction management aspect.

Generators or transformational systems take descriptions of an aspect (or an entire DBMS) as input and (semi-) automatically create corresponding implementations. Representatives of this approach are the EXODUS optimizer generator [33], its extension VOLCANO for parallel DBMSs [34], and GENESIS [2]. In GENESIS, a transformational approach is applied that maps one layer to the next lower one. This mapping has to be provided by the DBI.

In configuration approaches, the DBI selects reusable components, eventually modifies them, and plugs them together, resulting in a coherent subsystem or DBMS. Typical examples are the originally intended EXODUS approach [9], the transaction manager toolkit [56], and Open OODB [58]. Open OODB has been the first system that explicitly addressed the architectural aspect. It has a flexible architecture model, such that the DBMS-architecture can be tailored to the needs of the current construction. It also allows choices for several components ("policies"), but to the best of our knowledge there is neither support with respect to which policy to select, nor a mechanism that prevents incompatible choices.

Recently, a configuration-like approach has been proposed using so-called *data blade modules* for the object-relational DBMS Illustra [54] (which is now available as Informix Universal Server). A data blade module contains the definition and implementation of one or more abstract data types and can be plugged into Illustra. An examples for data blade module are geometric objects. Note that the data blade approach is a very convenient way to extend the functionality of a DBMS, while it does not support customization of the DBMS-implementation. Note finally that this approach relies on assumptions about the architecture and design of the DBMS into which data blades are plugged.

In comparison to the reviewed approaches and systems, the contributions of KIDS are the following:

- it is the first approach that aims at explicit support for a complete construction process,
- it strives for reusability in all phases (analysis, specification, design, and implementation),
- it supports an architecture model that (1) allows to reuse architectures and (2) is capable of integrating reused components,
- it is able to capture design alternatives thorugh the distinction of strategies and techniques,
- it supports different aspects and is open for further ones.

As mentioned in section 2, the KIDS project reuses experiences and techniques from software engineering. For the sake of brevity, we cannot give complete credits in this respect. Software engineering areas of great relevance are software reuse [39], research in software architectures [e.g., 47], and domain analysis [49].

7 Status and Future Work

The status of the KIDS project is as follows:

- A compiler for the DMSL language is operational. This compiler generates code that implements a functional (but operational) prototype of an object management subsystem according to a given data model specification.
- A compiler for IMSL is also available similar to the DMSL-compiler.
- A graphical editor and a parser for Spectra are also available. The repository for concurrency control strategies and techniques is still under construction as an O_2 [17] application.
- An extended version of the broker/services model is currently under construction [55].

In our future work, we intend to address the complex of software architectures and DBMSs in a broader sense. First, we are currently using (and extending) the broker/services model as well as our experiences in the KIDS project for DBMS-based process centered software development environments [55]. We also intend to use the broker/services model for modeling the software architecture of workflow management systems.

Second, while we are extending the repository of concurrency control strategies and techniques, we also want to use them in the two domains mentioned above. In contrast to the approach in KIDS, we then want to define concurrency control on a per-broker basis instead of having one broker being responsible for concurrency control of the whole system. We therefore will be concerned with using strategies to assist software architects in integrating different concurrency control mechanisms.

In conclusion, the KIDS project intends to produce a framework for the construction of DBMSs. It identifies two layers: a fixed one represented by the object server, and a variable one on top of the object server. A construction process model and an architecture model have been defined for the construction of that upper layer, and reusability has been of major concern for all phases and all aspects. In particular, we have shown that it is beneficial to determine customized reuse-based approaches for the various aspects/subsystems of DBMSs.

8 References

1. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, V. Watson. System R: Relational Approach to Database Management. *ACM Trans. on Database Systems* 1:2, 1976.

2. D.S. Batory, T.Y. Leung, T.E. Wise. Implementation Concepts for an Extensible Data Model and Data Language. *ACM Trans. on Database Systems* 13:3, 1988.
3. P.A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley 1987.
4. P.A. Bernstein, U. Dayal. An Overview of Repository Technology. *Proc. 20th Intl. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.
5. T.J. Biggerstaff, C. Richter. Reusability Framework, Assessment, and Directions. *IEEE Software*, July 1987.
6. A. Biliris. An Efficient Database Storage Structure for Large Dynamic Objects. *Proc. 8th Intl. Conference on Data Engineering*, Tempe, Arizona, February 1992.
7. A. Biliris, E. Panagos. A High Performance Configurable Storage Manager. *Proc. 11th Intl. Conf. on Data Engineering*, Taipei, Taiwan, March 1995.
8. R. Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments*. LNCS 562, Springer 1991.
9. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson. The Architecture of the EXODUS Extensible DBMS. In K.R. Dittrich, U. Dayal (eds): *Proc. 1986 International Workshop on Object-Oriented Database Systems*. IEEE Computer Science Press, 1986.
10. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. *Proc. 12th Intl. Conf. on Very Large Data Bases* (VLDB), Kyoto, Japan, August 1986.
11. M. Carey, L. Haas. *Extensible Database Management Systems*. *SIGMOD Record* 19:4, 1990.
12. H.-T. Chou, D.J. DeWitt, R.H. Katz, A.C. Klug. Design and Implementation of the Wisconsin Storage System. *Software - Practice and Experience* Vol. 15:10, 1985.
13. P.K. Chrysanthis. ACTA, A Framework For Modeling And Reasoning About Extended Transactions. PhD Thesis, Technical Report 91-26, Department of Computer Science, University of Pittsburgh, November 1991.
14. P.K. Chrysanthis, K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Trans. on Database Systems* 19:3, September 1994.
15. U. Dayal. Active Database Management Systems. *Proc. 3rd Intl. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, 1988.
16. B. Demuth, A. Geppert, T. Gorchs. Algebraic Query Optimization in the CoOMS Structurally Object-Oriented Database System. In J.C. Freytag, D. Maier, G. Vossen (eds): Query *Processing for Advanced Database Systems*. Morgan Kaufmann Publishers, 1993.
17. O. Deux. *The O2 System*. *Communications of the ACM* 34:10, 1991.
18. K.R. Dittrich, W. Gotthard, P.C. Lockemann. DAMOKLES - A Database System for Software Engineering Environments. R. Conradi, T.M. Didriksen, D.H. Wanvik (eds): *Advanced Programming Environments*. LNCS 244.
19. K.R. Dittrich, A.M. Kotz, J.A. Mülle. Database Support for VLSI-Design: The DAMASCUS System. M.H. Ungerer (ed): CAD-*Schnittstellen und Datentransferformate im Elektronik-Bereich*, Springer 1987.
20. K.R. Dittrich. Preface. In K.R. Dittrich (ed): *Advances in Object-Oriented Database Systems*. LNCS 334, Springer 1988.

21. K.R. Dittrich, S. Gatziu, A. Geppert (eds). The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In T. Sellis (ed), *Proc. 2nd Intl. Workshop on Rules in Database Systems*, Athens, Greece, September 1995. LNCS 985, Springer 1995.
22. R. Elmasri, S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing, 1989.
23. A. Finkelstein, J. Kramer, B. Nuseibeh (eds). *Software Process Modeling and Technology.* Research Studies Press Limited, 1994.
24. H. Garcia-Molina, K. Salem. Sagas. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.
25. D. Garlan, D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. *Proc. VDM '91: Formal Software Development Methods*, Noordwijkerhout, The Netherlands, October 1991. LNCS 551, Springer 1991.
26. A. Geppert, K.R. Dittrich. Rule-Based Implementation of Transaction Model Specifications. In N.W. Paton, H.W. Williams (eds): *Proc. Workshop on Rules in Database Systems* (RIDS), Edinburgh, UK, September 1993. Workshops in Computing, Springer-Verlag.
27. A. Geppert, K.R. Dittrich, V. Goebel, S. Scherrer. The NO2 Data Model. Technical Report 93.09, Institut fuer Informatik, Universitaet Zuerich, 1993.
28. A. Geppert, K.R. Dittrich. Constructing the Next 100 Database Management Systems: Like the Handyman or Like the Engineer? *ACM SIGMOD Record* 23:1, March 1994.
29. A. Geppert. Methodical Construction of Database Management Systems. Doctoral Dissertation, University of Zuerich, Switzerland, 1994.
30. A. Geppert, K.R. Dittrich. Strategies and Techniques: Reusable Artifacts for the Construction of Database Management Systems. In J. Iivari, K. Lyytinen, M. Rossi (eds): *Proc. 7th Intl. Conf. on Advanced information Systems Engineering* (CAiSE), Jyvaskyla, Finland, June 1995. LNCS 932, Springer 1995.
31. A. Geppert, S. Gatziu, K.R. Dittrich, H. Fritschi, A. Vaduva. Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS. Technical Report 95.29, Institut fuer Informatik, Universitaet Zuerich, November 1995.
32. J.A. Goguen, J. Messeguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In B. Shriver, P. Wegner (eds): *Research Directions in Object-Oriented Programming*. MIT Press Series in Computer Systems, Cambridge 1987.
33. G. Graefe, D.J. DeWitt. The EXODUS Optimizer Generator. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.
34. G. Graefe, W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *Proc. 9th Intl. Conf. on Data Engineering*, Vienna, Austria, April 19-23, 1993.
35. J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.
36. R.H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Washington, DC, May 1993.
37. T. Härder, A. Reuter. Concepts for Implementing a Centralized Database Management System. H.-J. Schneider (ed.): *Proc. Intl. Computing Symposium on Application Systems Development*, Nuernberg, Germany, March 1983.
38. M.F. Hornick, S.B. Zdonik. *A* Shared, Segmented Memory System for an Object-Orient-

ed Database. ACM Trans. on Office Information Systems 5:1, 1987.

39. C.W. Krueger. Software Reuse. *ACM Computing Surveys* 24:2, 1992.
40. B. Lindsay, J. McPherson, H. Pirahesh. A Data Management Extension Architecture. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.
41. B. Meyer. Applying "Design by Contract". *IEEE Computer* 25:10, 1992.
42. J.M. Neighbors. Draco: A Method for Engineering Reusable Software Systems. T.J. Biggerstaff, A.J. Perlis (eds): *Software Reusability. Volume I: Concepts and Models*. ACM Press, 1989.
43. S. Nittel. Concepts for Object Storage Servers in Configured Database Management Systems. Doctoral Dissertation, University of Zurich, October 1994 (in German).
44. P.E. O'Neil. The Escrow Transactional Method. *ACM Trans. on Database Systems* 11:4, 1986.
45. H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, U. Deppisch. Architecture and Implementation of the Darmstadt Database Kernel System. *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, CA, May 1987.
46. M.C. Paulk. The Evolution of the SEI's Capability Maturity Model for Software. *Software Process—Improvement and Practice*, Pilot Issue, 1995.
47. D.E. Perry, A.L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes* 17:4, 1992.
48. R. Prieto-Diaz, P. Freeman. Classifying Software for Reuse. *IEEE Software* 4:1,1987.
49. R. Prieto-Diaz, G. Arango (eds). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.
50. S. Scherrer. Specification and Prototypical Execution of Integrity Concepts for Domain-Specific Database Management Systems. Doctoral Dissertation, University of Zuerich, 1994.
51. S.S. Simmel, I. Godard. The Kala Basket. *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Phoenix, Arizona, October 1991.
52. V. Singhal, S.V. Kakkad, P.R. Wilson. Texas: An Efficient, Portable Persistent Store. *Proc. 5th Intl. Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.
53. R.A. Snowdon, R.C. Warboys. An Introduction to Process-Centered Environments. In [23].
54. M. Stonebraker, D. Moore. *Object-Relational DBMSs. The Next Great Wave*. Morgan Kaufmann Publishers, 1996.
55. D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C.B. Jones, C. Lengauer, H.-J. Schek (eds). *Object-Orientation with Parallelism and Persistence*. Kluwer Academinc Publishers, 1996
56. R. Unland, G. Schlageter. A Flexible and Adaptable Tool Kit approach for Transaction Management in Non Standard Database Systems. Informatik Berichte 109, Fernuniversitaet Hagen, Germany, June 1991.
57. G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. on Database Systems* 16:1, 1991.
58. D.L. Wells, J.A. Blakeley, C.W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer* 25:10.

59. J. Widom, R.J. Cochrane, B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. *Proc. 17th Intl. Conf. on Very Large Data Bases* (VLDB), Barcelona, Spain, September 1991.
60. K. Wilkinson, P. Lyngbaek, W. Hasan. The Iris Architecture and Implementation. *IEEE Trans. on Knowledge and Data Engineering* 2:1, 1990.