Chapter 4

Architecture Synthesis Process

4.1 Introduction

Research on software architecture design approaches is still in its progressing phase and several architecture design approaches have been introduced in the last years [Bass et al. 98], [Buschmann et al. 99], [Tracz & Coglianese 92], [Shaw 98]. However, a consensus on the appropriate software architecture design process is not established yet and current software architecture design approaches may have to cope with several problems.

First of all, planning the architecture design phase is intrinsically difficult due to its conflicting goals of providing a gross level structure of the system and at the same time directing the subsequent phases in the project. The first goal requires planning the architecture in later phases of the software development process when more information is available. The latter goal requires planning it as early as possible so that the project can be more easily managed.

Second, most software architecture design approaches derive the architectural abstractions in different ways and from different sources such as artifacts, use-cases, patterns and problem domains. These sources are basically focused on the client's-perspective[1] rather than on the architectural solution perspective of the system. The gap between the client perspective and the architectural design perspective is generally too large and the client may lack to specify the right detail of the problem, thereby either under-specifying or over-specifying the problem. This on its turn hinders the identification of the right architectural abstractions since the fundamental transparent abstractions may be missed or redundant abstractions may be elicited.

Third, generally the adopted sources are also not very useful to provide sufficiently rich semantics of the architectural components and fall short in providing guidelines for composing the architectural abstractions. In this case, architectural components are often equivalent to semantically poor groupings of artifacts and are composed using simple associations.

Finally, although solution domain analysis may be used and be effective in deriving the architectural abstractions and provide the necessary semantics, it may not suffice if it is not managed well. The problem is that the domain model may lack the right detail of abstraction to be of practical use for deriving architectural abstractions.

Current architecture design approaches have to cope with one or more of the above problems. In this chapter, a novel approach is proposed, which is termed as *synthesis-based software architecture design* that aims to provide effective solutions to these problems. *Synthesis* is a well-known concept in

[1] We use the term client to denote any stakeholders who has interest in the application of a software architecture.

traditional engineering disciplines and involves the construction of sub-solutions for distinct loosely coupled sub-problems and the integration of these sub-solutions into a complete solution. During the synthesis process design alternatives are searched and selected based on the existing solution domain knowledge.

In the synthesis-based software architecture design approach, the synthesis concept of traditional engineering disciplines is applied to the software architecture design process. Hereby, the requirements are first mapped to technical problems. For each problem the corresponding solution domain is identified and architectural abstractions are derived from the solution domain knowledge. The solution domain knowledge provides well-established concepts with rich semantics and as such form a stable basis for architecture development. The individual sub-solutions are combined in the overall software architecture.

In this chapter we will demonstrate the approach using a project on the design of an atomic transaction system architecture for a distributed car dealer information system[2].

The remainder of the chapter is organized as follows. In section 4.2, the synthesis concept is described and a model for software architecture synthesis is derived. In section 4.3, an example project on the design of a software architecture for atomic transactions for a distributed car dealer information system will be described. This example project will be used throughout the whole chapter. In section 4.4, the synthesis-based architecture design approach will be presented that will be illustrated for the example project. Finally, in section 4.5, we will present our discussion and conclusions.

4.2 Synthesis

This section describes the concept of synthesis. *Synthesis* is a well-known concept in traditional engineering disciplines and is widely applied to solve design problems [Maher 90]. Software architecture design can be considered as a problem solving process in which the problem represents the requirement specification and the solution represents the software architecture design. In this section we apply the synthesis process to the software architecture design process. In section 4.2.1 we will explain the concept *synthesis* as it is described in traditional engineering disciplines. In section 4.2.2 we will apply the concept synthesis to software architecture design and gradually derive the steps for defining the software architecture synthesis model.

[2] This has been carried out as part of the INEDIS project that was a cooperative project between Siemens-Nixdorf and the TRESE group, Software Engineering, Dept. of Computer Science, University of Twente.

4.2.1 Synthesis in Traditional Engineering

Synthesis in engineering often means a process in which a problem specification is transformed to a solution by first decomposing the problem into loosely coupled sub-problems that are independently solved and integrated into an overall solution.

Synthesis consists generally of multiple steps or cycles. A synthesis cycle corresponds to a transition (transformation) from one *synthesis state* to another and can be formally defined as a tuple consisting of a problem specification state and a design state [Maimon & Braha 96]. The problem specification state defines the set of problems that still needs to be solved. The design state represents the tentative design solution that has been lastly synthesized. Initially, the design state is empty and the problem specification state includes the initial requirements. After each synthesis state transformation, a sub-problem is solved. In addition a new sub-problem may be added to the problem specification state.

Each transformation process involves an evaluation step whereby it is evaluated whether the design solutions so far (design state) are consistent with the initial requirements and any additional requirements identified during the synthesis.

A *synthesis-based design process* is defined as a finite sequence of synthesis states, resulting in a terminal state. A synthesis state is terminal in either of two cases: the specification part is satisfiable by the design part (there is a solution) or neither the design nor the specification can be modified. The first is a successful design the latter is an unsuccessful one.

The sub-solutions and overall solution has to meet a set of objective metrics, while satisfying a set of constraints. Constraints may be imposed within and among the sub-solutions. For a suitable synthesis it is required that the problem is understood well. This means that the problem is well-described and the quality criteria and constraints are known on beforehand. In practice, however, this is very difficult to meet and complete analysis is impossible in any but the simplest problems [Coyne et al. 90]. Therefore, synthesis can usually start before the problem is totally understood.

During the synthesis process a designer needs to consider the design space that contains the knowledge that is used to develop the design solution. For this, synthesis requires the ability to produce a set of alternative solutions and select an optimal or near optimal solution. The space of possible solutions, however, may be very large and it is not feasible to examine all possible solutions [Coyne et al. 90].

In [Maimon & Braha 96] it has been shown that the design synthesis is inherently an NP-Complete problem. To manage this inherent complexity, synthesis can be performed at different, higher abstraction levels in the design process. In the design of digital signal processing systems, for example, the following synthesis approaches with increasing abstraction levels are distinguished: circuit synthesis, logic synthesis, register-transfer synthesis, and system synthesis [Gajski et al. 92].

For large problems, the lower-level design synthesis approaches become intractable and time consuming due to the large number of entities and their relations that need to be considered. In the example of digital signal processing, circuit synthesis adopting the transistor as the basic abstraction, is unsuitable for current industrial problems that integrate millions of components. A higher level of abstraction reduces the number of entities that a designer has to consider which in turn reduces the complexity of the design of larger systems. In addition, higher level abstractions are closer to a designer's way of thinking and as such increases the understandability, which on its turn facilitates to consider various alternatives more easily. The counterpart is that higher level abstractions consists of the fixed configuration of lower level abstractions thereby reducing the alternative configuration possibilities, that is, the set of alternatives is implicitly reduced. This is acceptable, though, since usually the total space of a synthesis from higher level abstractions is large enough to be of practical use.

4.2.2 Defining the Software Architecture Synthesis Model

Mapping Client Requirements to Technical Problems

Client requirements may lack to specify the right detail of the problem and either under-specify or over-specify the problem domain. Therefore, the gap between the requirements and the architectural design solution is generally too large. To solve this problem we propose to introduce a *problem analysis* phase that functions as an intermediary process between the requirements analysis and architectural design. Within this problem analysis phase, the delivered client requirements are thoroughly analyzed and mapped to technical problems that describe the problems more accurately. In this way, the gap between the requirements and the architectural design is largely reduced and, once the problems are clearly understood and specified independently of the initial requirements, a solid basis is provided to drive the architecture development.

Figure 1 illustrates the separation of the concept *Technical Problem* from the concept *Requirements*. The rounded rectangles represent the concepts; the directed arrows represent the functions between the concepts. The left side of the figure before the hollow arrow represents the approach that is adopted in several architecture design approaches. Hereby, the requirements are basically directly mapped to the (architectural) solution abstractions. The right side of the figure represents the introduction of the concept *Technical Problem* that has been separated from the concept *Requirements*. Hereby the concept *Requirements* is not directly mapped to the concept *Solution* but first it is analyzed and mapped to the concept *Technical Problem*. The concept *Technical Problem* describes the fundamental aspects that may not have been present in the original requirements. A clear understanding of the problem is part of the solution and as such this reduces the distance to the final solution.

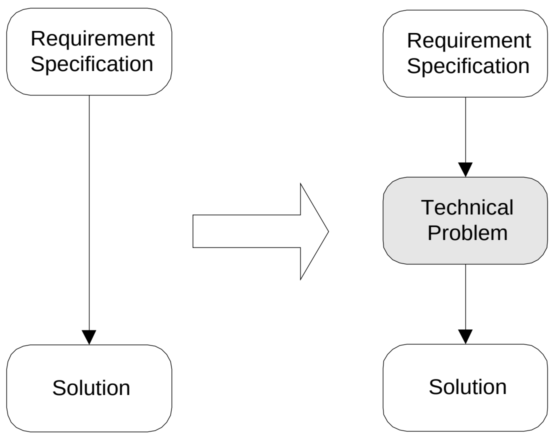

Figure 1. The separation of the concept *Technical Problem* from the concept *Requirements*

Deriving Architectural Abstractions from Solution Domain Models

We maintain that architectural abstractions should be best derived from the solution domain knowledge. The reason for this is threefold:

First, the solution domain knowledge includes well-established concepts that will not change abruptly. This is because solution domain knowledge is defined by a thorough analysis and research and is sufficiently stabilized through a consensus of experts in the corresponding community. A basic requirement for architectural components is that they should be stable and solution domain concepts provide this stability.

Second, the solution domain concepts are semantically rich, and define the properties, the relations with other concepts and their behavior. As such solution domain concepts may provide the architectural components also the required rich semantics.

Third, solution domain concepts are related to each other and structured into taxonomies and partimonies. Further, the compatibility and composition relations between the various concepts are also well-defined. Existing architectural design approaches provide weak support for composing the architectural components as we have described in chapter 3. Solution domain knowledge provides the necessary information to support the composition of architectural components.

There exist domain analysis approaches that aim to provide solution domain models [Prieto-Diaz & Arrango 91] [Arrango 94] [Wartik & Prieto-Diaz 92]. We argue that these approaches should be integrated within the architecture design approaches to derive stable architectures.

Leveraging Solution Domain Models to the Identified Technical Problems

The solution domain analysis should be appropriately managed so that the domain model is optimally tuned to the architectural design phase. The right level of detail of the solution domain model can only be defined if both the client’s requirements and the corresponding solution domain are considered. On the one hand, the initial client requirements will likely fail to accurately define the overall-scope and the relevant abstractions because it does not provide a solution perspective of the problem. On the other hand, the solution domain itself may be very large and include abstractions that are not relevant for solving the corresponding problem. We maintain that the separated problem specifications from the client requirements provide a useful basis for leveraging the solution domain knowledge. This is because it is supposed to describe all the necessary fundamental aspects for solving the problem.

This requirement is illustrated in Figure 2, which is a refinement of Figure 1. The refinement here consists of the introduction of the concept *Solution Domain Knowledge.*

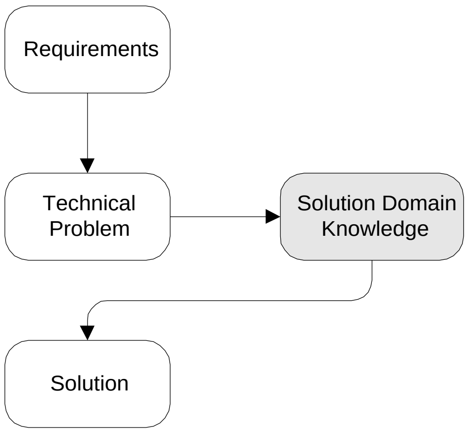

Figure 2. Leveraging solution domain knowledge to the problems

The arrow from the concept *Solution Domain Knowledge* to the concept *Solution* represents the previous requirement of deriving solution abstractions from solution domain knowledge. The arrow from the concept *Technical Problem* to the concept *Solution Domain Knowledge* represents the search and leveraging of the solution domain knowledge by the identified problems.

Defining Architecture Iteratively and Recursively

Planning the architectural design phase is intrinsically difficult due to the conflicting goals of its intended use. On the one hand it needs to represent the gross-level structure of the system and for this it is necessary to have a complete overview of the system including the analysis and design models in the later phases of the software development process. In addition, it is intended to be used to manage the project adequately and this requires defining the architecture as early as possible.

We require that architectures should be derived from solution domain knowledge that we proposed to leverage according to the identified problems as it has been illustrated in Figure 2.

To solve this dilemma we argue to adopt both an iterative and a recursive architecture design approach. Iteration means that the same steps of a process are repeated to correct what has already been done. Recursion means that the same steps of a process are repeated for a lower abstraction level. For architecture design approach iteration means that the process is repeated to correct the architectural abstractions because of newly acquired information. Recursion means that the same process is repeated to define the sub-architectural concepts. The process of iteration and recursion is visualized in Figure 3.

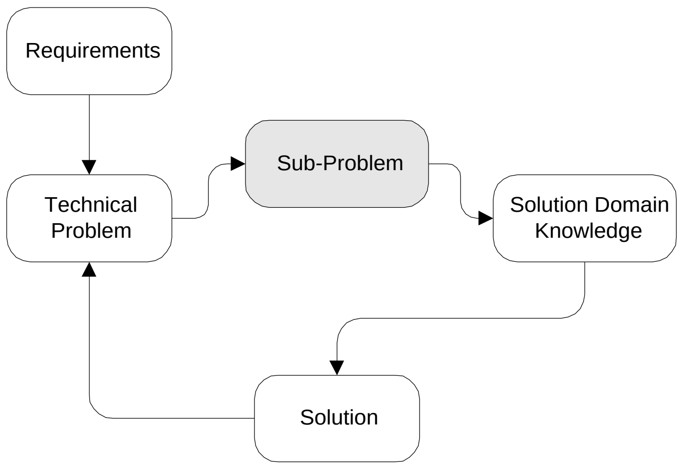

Figure 3. Recursion and Iteration in providing Architectural Design Solution

This figure is a refinement of Figure 2 and introduces the new concept *Sub-Problem*. The recursion process is basically defined by the decomposition of the problem into sub-problems whereby the suitable concepts are searched from the solution domain for each sub-problem individually. The iteration is represented by the arrow directed from the concept *Solution* to the concept *Problem*.

The Software Architecture Synthesis Model

Figure 4 represents a model of architecture design synthesis[3]. The model consists of two parts: *Solution Definition* and *Solution Control*. Each part consists of concepts and functions among concepts. The concepts are represented by rounded rectangles, the functions are represented by arrows. The part *Solution Definition* represents the identification and definition of solution abstractions. The part *Solution Control* represents the quantification, measurement, optimization and refinement of the selected solution abstractions. In the following we will explain the concepts and functions of both parts of the model.

[3] Note that this model conforms to the CPC model described in chapter 2.

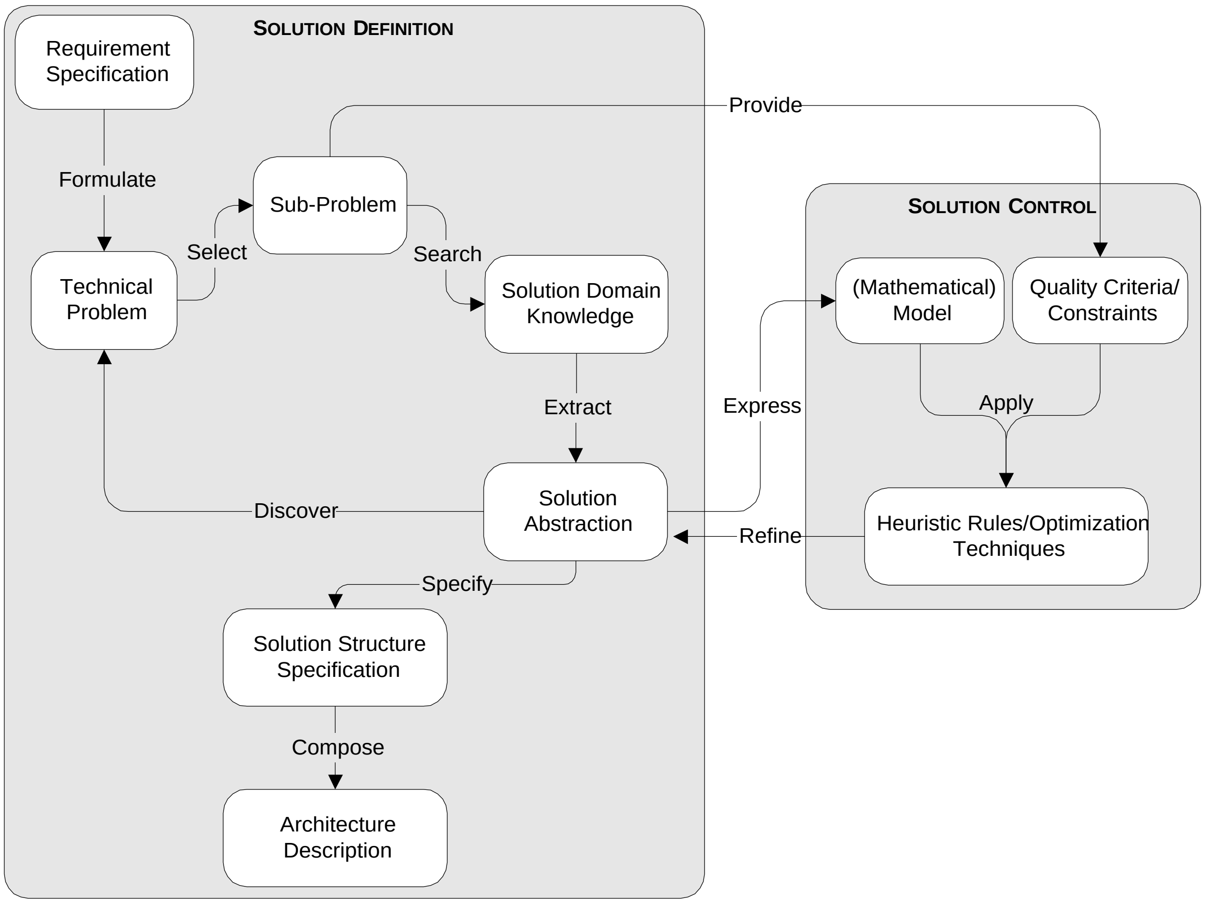

Figure 4. The Architecture Synthesis Model

Solution Definition

The concept *Requirement Specification* represents the requirements of the stakeholders who are interested in the development of a software architecture.

The concept *Technical Problem* represents the problem specification that is actually to be solved. The model thus separates the concepts *Requirement Specification* and *Technical Problem*.

The function *Formulate* defines the process for searching and representing the problems that need to be solved for the architecture development.

The concept *Sub-Problem* represents a sub-problem of the identified problem.

The function *Select* represents the process for selecting the corresponding sub-problem from the problem.

The concept *Solution Domain Knowledge* represents the solution domain knowledge that is needed for solving the sub-problem.

The function *Search* represents the process for searching the solution domain knowledge for a given problem.

The concept *Solution Abstraction* represents the extracted solution from the solution domain knowledge.

The function *Extract* represents the process for extracting the solution abstractions from the solution domain knowledge.

The concept *Solution Structure Specification* represents the specification of the extracted solution abstraction.

The function *Specify* represents the process for specifying the solution abstraction.

The concept *Architecture Description* represents the architecture description so far.

The function *Compose* represents the refinement of the overall-architecture description with the concept *Solution Structure Specification.*

The function *Discover* represents the process of discovering new sub-problems when new solution abstractions are extracted from the solution domain knowledge.

Solution Control

The part *Solution Control* has conceptual relations with the part *Solution Definition* through the functions *Provide*, *Express* and *Refine.*

The function *Provide* represents the process for providing the quality criteria and constraints that are imposed on the solution. The concept *Quality Criteria/Constraints* represents these criteria and constraints of the (sub-) problem.

The function *Express* represents a formalization of the solution abstraction for evaluation purposes. Typical formalizations may be the quantification into mathematical models.

The function *Apply* represents the process for measurement of the expressed solution abstraction using the provided quality criteria / constraints.

The concept *Heuristic Rules / Optimization Techniques* represents the optimization of the formalizations of the solution abstractions. It can be based on mathematical optimization techniques or heuristic rules.

The function *Refine* represents the process of refining the solution abstraction according to the results of the optimization techniques.

4.3 Example Project: Transaction Software Architecture Design

The Integrated New European Dealer Information System project (INEDIS) has been carried out as a collaborative project between the TRESE group of the University of Twente and Siemens-Nixdorf, The Netherlands. The project dealt with the development of a distributed car dealer information system in which different car dealers are connected through a network. The system needs to provide automated support for processes such as workshop processing, order processing, stock management, *n*ew and used car management, and financial accounting. The car dealer system should execute the provided tasks consistently and effectively. The meaning of consistency depends on any a priori constraints, which must be guaranteed that they will never be violated. For example, two clients may not reserve the same car at the same time.

There are two main factors that threaten the consistency of data in a distributed system: *concurrency* and *failures*. In case of concurrency the executions of programs that access the same objects can interfere. When a failure occurs, one or more application programs may be interrupted in midstream. Since a program is written under the assumption that its effects are only correct if it would be executed in its entirety, an interrupted program may lead to inconsistencies as well. To achieve data consistency distributed systems should include provision for both concurrency and recovery from failures. The implementation of these concurrency and recovery mechanisms, however, should be transparent to the application program developers, since they will need only the primitives and don't want to be bothered with implementation details. *Atomic transactions*, or simply transactions, are a well-known and fundamental abstraction which provide the necessary concurrency control and recovery mechanisms for the application programs in a transparent way. Transactions relieve application programmers of the burden of considering the effects of concurrent access to objects or various kinds of failures during execution. Atomic transactions have proven to be useful for preserving the consistency in many applications like airline reservation systems, banking systems, office automation systems, database systems and operating systems.

The car dealer information system also required the use of atomic transactions. The system would be used in different countries and by different dealers each requiring dedicated transaction protocols. Therefore, a basic requirement of the system was to identify common patterns of transaction systems and likewise provide a stable architecture of atomic transactions that could be customized to the corresponding needs.

Next to the need for adaptability at initialization time the system required also adaptation at run-time. The system may be constituted of a large number of applications with various characteristics, operates in heterogeneous environments, and may incorporate different data formats. To achieve optimal behavior, this requires transactions with dynamic adaptation of transaction behavior, optimized with respect to the application and environmental conditions, and data formats. The adaptation policy, therefore, must be determined by the programmers, the operating system or the data objects. Further, reusability of the software is considered as an important requirement to reduce development and maintenance costs.

4.4 Synthesis-based Software Architecture Design

In this section the synthesis-based software architecture design process that implements the process of the Architecture Synthesis Model of Figure 4 will be described. This approach is illustrated in Figure 5.

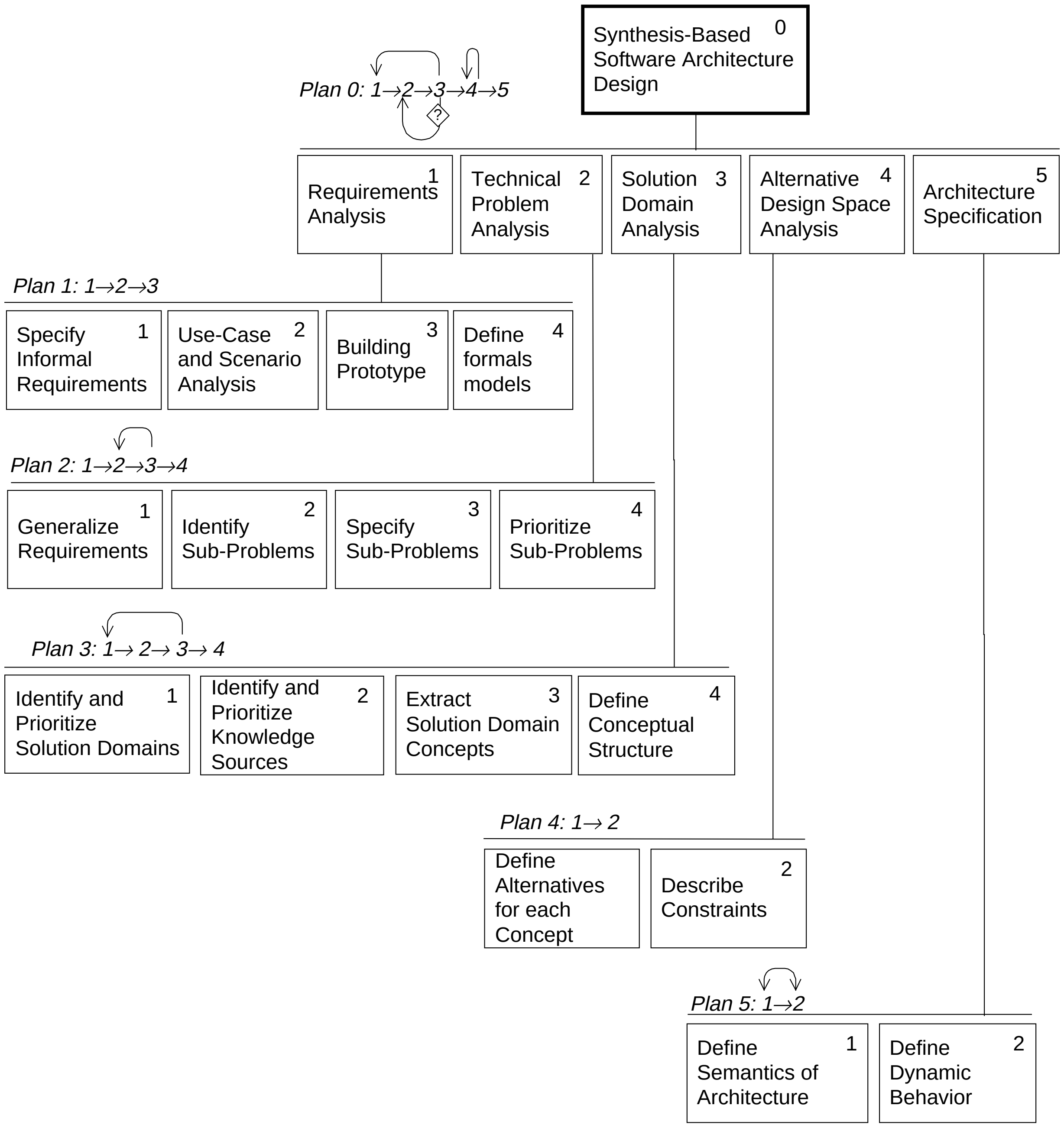
Synthesis-Based Software Architecture Design 0
Plan 0: 1→2→3→4→5
?
1 Requirements Analysis
Technical Problem Analysis 2
Solution Domain Analysis 3
Alternative Design Space Analysis 4
5 Architecture Specification
Plan 1: 1→2→3
Specify Informal Requirements 1
Use-Case and Scenario Analysis 2
3 Building Prototype
Define formals models 4
Plan 2: 1→2→3→4
1 Generalize Requirements
2 Identify Sub-Problems
3 Specify Sub-Problems
4 Prioritize Sub-Problems
Plan 3: 1→ 2→ 3→ 4
Identify and Prioritize Solution Domains 1
Identify and Prioritize Knowledge Sources 2
Extract Solution Domain Concepts 3
Define Conceptual Structure 4
Plan 4: 1→ 2
Define Alternatives for each Concept
2 Describe Constraints
Plan 5: 1→2
Define Semantics of Architecture 1
Define Dynamic Behavior 2

Figure 5. Synthesis-based Software Architecture Design Approach

The figure uses the graphical notation from Hierarchical Task Analysis (HTA) [Diaper 89b] in which activities are represented in hierarchical order. Each numbered box represents an activity that can be refined using a plan. Each plan represents a flow diagram describing the causal sequencing of the activities. The double-headed arrows represent interaction between two activities. The diamond with a question mark represents the validation of a step.

The following sections are organized around the basic process of the approach. Section 4.4.1 describes the *Requirements Analysis process*, section 4.4.2 the *Problem Analysis process*, section 4.4.3 the *Solution Domain Analysis process*, section 4.4.4 *Alternative Space Analysis process* and finally section 4.4.5 the *Architecture Specification process*.

4.4.1 Requirements Analysis

The architecture design is initiated with the requirements analysis phase in which the basic goal is to understand the stakeholder requirements. Stakeholders may be managers, software developers, maintainers, end-users, customers etc. [Prieto-Diaz & Arrango 91]. In the synthesis-based approach the well-known requirement analysis techniques such as textual requirement specifications, use-cases [Jacobson et al. 99] and scenarios [Kruchten 95], constructing prototypes and defining finite state machine modeling are used. Informal requirement specifications serve as a first basis for the requirements analysis process and is generally defined by interacting with the clients. Use cases provide a more precise and broader perspective of the requirements by specifying the external behavior of the system from different user perspectives. Scenarios are instances of use cases and define the dynamic view and the possible evolution of the system. Prototypes are used to define the possible user interfaces and may further help to clarify the desired behavior of the system. Finally, for safety-critical systems rigorous approaches such as state transition diagrams or formal specification languages may be used.

These techniques have been applied in different approaches and have shown to be useful in supporting the analysis and understanding of the client requirements. We will not elaborate on them in this thesis and refer for detailed information to the corresponding publications [Thayer et al. 97] [Sommerville & Sawyer 97] [Loucopoulos & Karakostas 95].

Example

At the start of the project the initial requirement specification was given by the client [Ahsmann & Bergmans 95]. We interviewed developers and managers of the project to extract the basic requirements for the INEDIS system [Tekinerdogan 95a]. We further analyzed the project literature, which included user’s guide, manuals, design and implementation documentation and case studies. In addition we experimented with the existing NEDIS system in a real environment and identified the basic requirements for the further releases. Thereby, we were accompanied by the developer and maintainers of the system. Next to the overall requirements of the INEDIS system we focused on the requirements that were specific for atomic transactions [Tekinerdogan 95b][Tekinerdogan 96].

From this study we were able to set up the basic requirements that we expressed in use cases. Figure 6 represents the use case model for transaction processing. It has one actor, Dealer, and four use cases, namely *initiate transaction, start transaction, abort transaction,* and *commit transaction*. The use case *initiate transaction* will be performed for describing and preparing a program to be used as a transaction. The use case *start transaction* will invoke the operations to access the transaction objects. Finally, the use cases *abort transaction* and *commit transaction* will describe the abort respectively the commit actions.

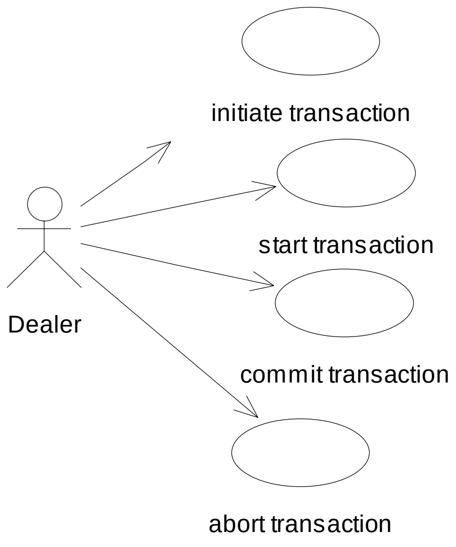

Figure 6. A use case model for the transaction processing in the INEDIS system

For a more detailed requirements analysis we refer to the project's requirements analysis documents [Tekinerdogan 95a][Tekinerdogan 95b][Tekinerdogan 96].

4.4.2 Technical Problem Analysis

The requirements analysis process provides an understanding of the client perspective of the software system. As it is described in Figure 5, the next step involves the technical problem analysis process in which client requirements are mapped to technical problems. This is to say that the architecture design process is to be considered as a problem solving process in which the solution represents an architecture design. The problem analysis process consists of the following steps:

1. *Generalize the Requirements:* whereby the requirements are abstracted and generalized.

2. *Identify the Sub-Problems*: whereby technical problems are identified from the generalized requirements.
3. *Specify the Sub-Problems*: whereby the overall technical problem is decomposed into sub-problems.
4. *Prioritize the Sub-Problems*: whereby the identified technical problems are prioritized before they are processed.

Let us explain these processes in more detail now.

Generalize the requirements

Discovering the problems from a requirement specification is not a straightforward task. The reason for this is that the clients may not be able to accurately describe the initial state and the desired goals of the system. The client requirements may be specific and provide only specific wordings of a more general problem. Therefore, to provide the broader view and identify the right problems we abstract and generalize from the requirement specification and try to solve the problem at that level[4]. Often, this abstraction and generalization process allows to define the client's wishes in entirely different terms and therefore may suggest and help to discover problems that were not thought of in the initial requirements.

Identify sub-problems

Once the requirement specification has been put into a more general and broader form, we derive the technical problem that consists usually of several sub-problems. At this phase, architecture design is considered as a problem solving process. Problem solving is defined as the operation of a process by which the transformation from the initial state to the goal is achieved [Newell & Simon 76]. We need thus first to discover and describe the problem. For this, in the generalized requirement specification we look for the important aspects that needs to be considered in the software architecture design [Tekinerdogan & Aksit 99]. These aspects are identified by considering the terms in the generalized requirements specification, the general knowledge of the software architect and the interaction with the clients. This process is supported by the results of the requirements analysis phase and utilizes the provided use-case models, scenarios, prototypes and formal requirements models.

[4] In mathematics, solving a concrete problem by first solving a more general problem is termed as the *Inventor's Paradox* [Polya 57] [Lieberherr 96]. The paradox refers to the fact that a general problem has paradoxically a simpler solution than the concrete problem.

Specify sub-problems

The identification of a sub-problem goes in parallel with its specification. The major distinction between the identification and the specification of a problem is that the first activity focuses on the process for finding the relevant problems, whereas the second activity is concerned with its accurate formalization. A problem is defined as the distance between the initial state and the goal. Thereby, the specification of the technical problems consist of describing its name, its initial state and its goal.

Prioritize sub-problems

After the decomposition of the problem into several sub-problems the process for solving each of the sub-problems can be started. The selection and ordering in which the sub-problems are solved, though, may have an impact on the final solution. Therefore, it is necessary to prioritize and order the sub-problems and process the sub-problems according to the priority degrees. The prioritization of the sub-problems may be defined by the client or the solution domain itself. The latter may be the case if a sub-problem can only be solved after a solution for another sub-problem has been defined.

Example

We generalized the INEDIS requirement specification [Ahsmann & Bergmans 95] and mapped these to the technical problems. For example, we generalized the requirements for the various scheduling techniques. In the original requirement specification and the interview with the stakeholders we identified that only two concurrency control approaches were used, namely optimistic and aggressive locking. Attempts were made to adapt between these two concurrency control mechanisms. After our discussion with the stakeholders [Tekinerdogan 95b] it followed that the system needed also other types of concurrency control protocols and the run-time adaptation had to be defined for these as well.

- *P0*

Name: Provide adaptable architecture of atomic transactions
Initial State: This is the overall problem. Initially no transaction architecture design was available.
Goal: Identify the fundamental abstractions of transactions and design the atomic transaction software architecture that can be reused for different dealers and includes dynamic adaptation mechanisms for the different transaction protocols.

In parallel with our generalization of the requirements we were able to define the different sub-problems, which are listed in the following:

- *P1*

Name: Provide transparent concurrency control.

Initial State: Limited concurrency control techniques.

Goal: Determine the set of concurrency control techniques that are required and provide this in a reusable form.

- ***P2***

Name: Provide transparent recovery techniques.

Initial State: Limited recovery techniques based on simple data types.

Goal: Determine the set of recovery techniques that can be used for various kinds of data types and provide this in a reusable form.

- ***P3***

Name: Provide transparent transaction management techniques.

Initial State: Transaction management is primitive and is based on flat transactions only. The Start, Commit and Abort protocols are fixed.

Goal: Provide various transaction management techniques that can be applied for advanced transactions such as long transactions and nested transactions. Provide the various start, commit and abort protocols in a reusable format.

- ***P4***

Name: Provide adaptable transaction protocols based on transaction, system and data criteria.

Initial State: Selection of transaction protocols such as transaction management, concurrency control and recovery protocol is fixed.

Goal: Provide the means to adapt the transaction protocols both on compile-time and run-time. Adaptation mechanism should be determined by programmers, operating system or the data object characteristics.

After interactive discussions with the stakeholders the above sub-problems have been prioritized in the given order, thus, P1, P2, P3 and P4. Figure 7 represents the *problem structure diagram*. In the problem structure diagram the nodes represent the technical problems and the lines the nesting relations. The nodes are numbered according to their nesting level. The problem structure diagram helps to sharpen and improve the understanding of the problem and can be used to reach a consensus with the client on the addressed problems. Note that the problem structure diagram in Figure 7 includes also the sub-problems of the problems that we described above. These sub-problems have been only identified during the refinement process after the solution domain analysis process. We will explain these in later sections. From this it follows that the problem structure diagram is not static but probably changes during the architecture design process because the problem may not be analyzed from a complete isolation from the solution domain [Cross 89].

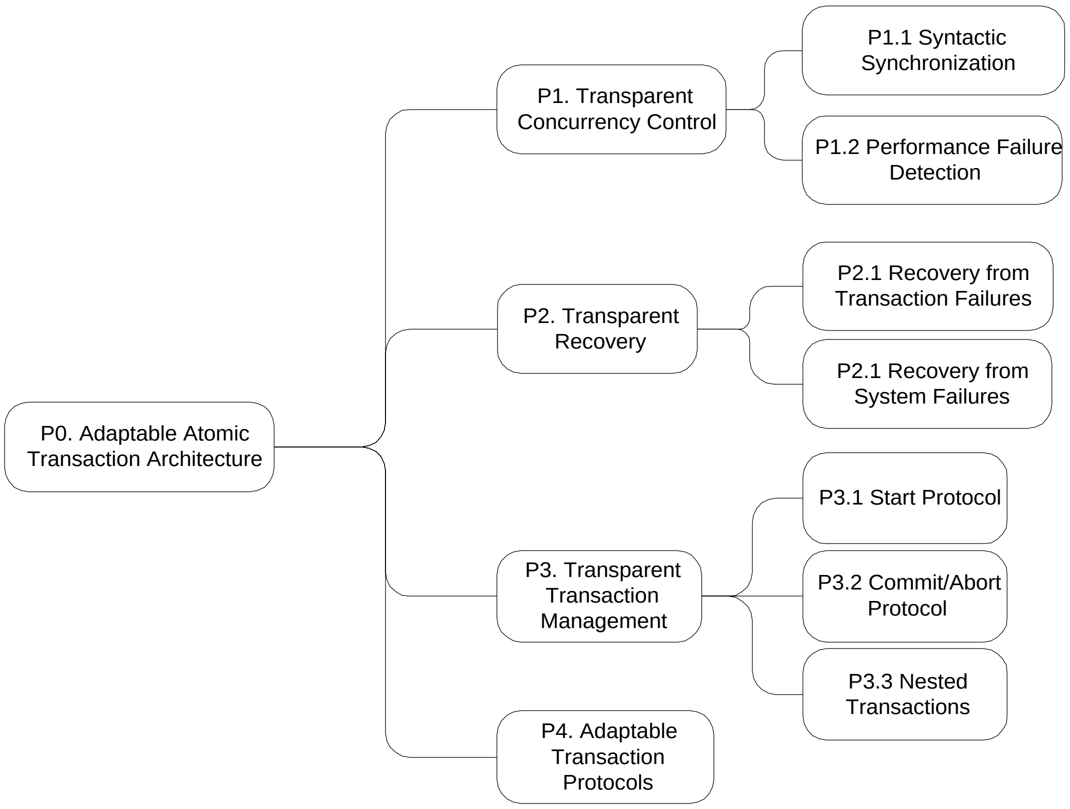
P0. Adaptable Atomic Transaction Architecture
P1. Transparent Concurrency Control
P1.1 Syntactic Synchronization
P1.2 Performance Failure Detection
P2. Transparent Recovery
P2.1 Recovery from Transaction Failures
P2.1 Recovery from System Failures
P3. Transparent Transaction Management
P3.1 Start Protocol
P3.2 Commit/Abort Protocol
P3.3 Nested Transactions
P4. Adaptable Transaction Protocols

Figure 7. Problem structure diagram for the example project

4.4.3 Solution Domain Analysis

The Solution Domain Analysis process aims to provide a solution domain model that will be utilized to extract the architecture design solution. It consists of the following activities:

1. *Identify and prioritize the solution domains for each sub-problem*
2. *Identify and prioritize knowledge sources for each solution domain.*
3. *Extract solution domain concepts from solution domain knowledge.*
4. *Structure the solution domain concepts.*
5. *Refine the solution domain concepts.*

In the following we will explain these sub-processes. In Figure 5, the first four activities are represented from plan 3.1 to plan 3.4. The refinement of the solution domain concepts is represented by a directed arrow from plan 0.3 to plan 0.1.

To understand the relations between these activities Figure 8 represents a conceptual model for illustrating the relations between the concepts *Technical Problem*, *Sub-Problem*, *Solution Domain*, and *Solution Domain Concept*.

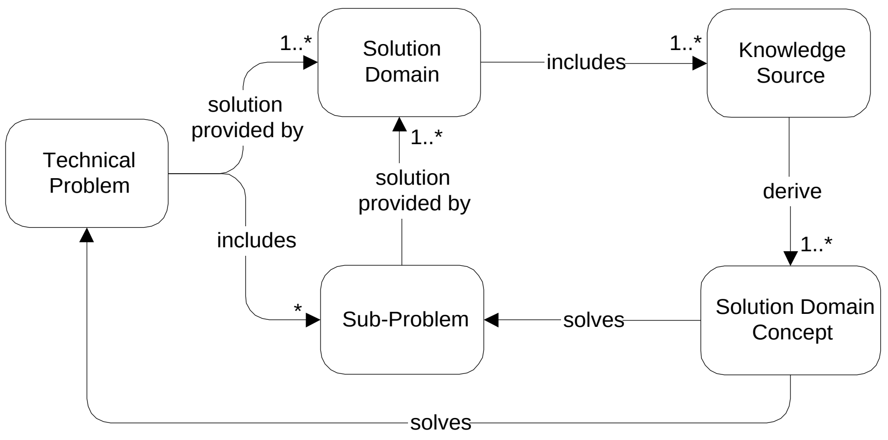

Figure 8. The relations from Problem to Solution Domain Concept

Hereby, the rounded rectangles represent the concepts and the directed arrows represent the associations between these concepts. From the figure it follows that for each *Technical Problem* a solution is provided by one or more *Solution Domains*. The concept *Problem* includes zero or more *Sub-Problems*. Each *Solution Domain* includes 1 or more *Knowledge Sources* from which 1 or more *Solution Domain Concepts* may be derived that solves the concepts *Problem* and *Sub-Problem*.

Identify and Prioritize the Solution Domains

For the overall problem and each sub-problem we search for the solution domains that provide the solution abstractions to solve the technical problem. The solution domains for the overall problem are more general than the solution domains for the sub-problems. In addition, each sub-problem may be recursively structured into sub-problems requiring more concrete solution domains on their turn.

An obstacle in the search for solution domains may be the possibly large space of solution domains leading to a time-consuming search process. To support this process, we look for categorizations of the solution domain knowledge into smaller sub-domains. There are different categorization possibilities [Glass & Vessey 95]. In library science, for example, the categories are represented by *facets* that are groupings of related terms that have been derived from a sample of selected titles [Rubin 98]. In [Aksit 00], the solution domain knowledge is categorized into *application*, *mathematical* and *computer science* domain knowledge. The application domain knowledge refers to the solution domain knowledge that defines the nature of the application, such as reservation applications, banking applications, control systems etc. Mathematical solution domain knowledge refers to mathematical knowledge such as logic, quantification and calculation techniques, optimization techniques, etc. Computer science domain refers to knowledge on the computer science solution abstractions, such as programming languages, operating systems, databases, analysis and design methods etc. This type of knowledge has been recently compiled in the so-called Software Engineering Body of Knowledge (SWEBOK) [Bourque et al. 99]. Notice that our approach does not favor a particular categorization of the solution domain knowledge and likewise other classifications besides of the above two approaches may be equally used.

If the solution domains have been adequately organized one may still encounter several problems and the solution domain analysis may not always warrant a feasible solution domain model. This is especially the case if the solution domains are not existing or the concepts in the solution domain are not fully explored yet and / or compiled in a reusable format. Figure 9 shows the flow diagram for the feasibility study on solution domain analysis. Hereby, the diamonds represent decisions, the rectangles the processes and the rounded rectangle the termination of the flow process.

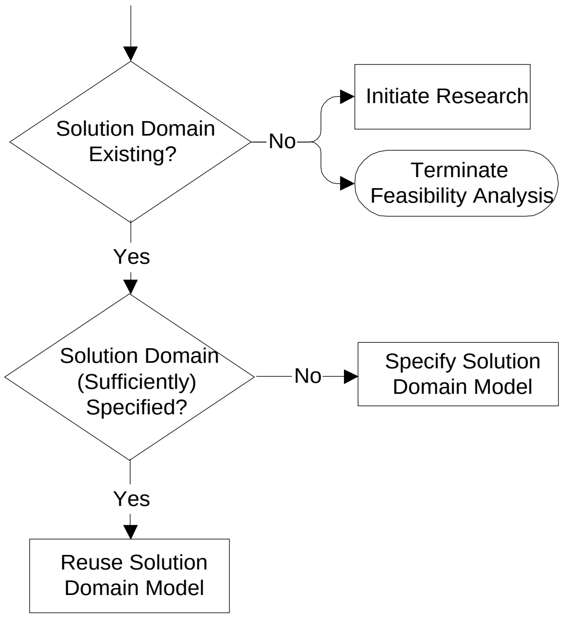
Solution Domain Existing?
No
Initiate Research
Terminate Feasibility Analysis
Yes
Solution Domain (Sufficiently) Specified?
No
Specify Solution Domain Model
Yes
Reuse Solution Domain Model

Figure 9. Flow diagram for feasibility study on solution domain analysis

If the solution domain knowledge is not existing, one can either terminate the feasibility analysis process or initiate a scientific research to explore and formalize the concepts of the required solution domain. The first case leads to the conclusion that the problem is actually not (completely) solvable due to lack of knowledge. The latter case is the more long-term and difficult option and falls outside the project scope.

If a suitable solution domain is existing and sufficiently specified, it can be (re)used to extract the necessary knowledge and apply this for the architecture development. It may also happen that the solution domain concepts are well-known but not formalized [Shaw & Garlan 96]. In that case it is necessary to specify the solution domain.

Identify and Prioritize Knowledge Sources

Each identified solution domain may cover a wide range of solution domain knowledge sources. These knowledge sources may not all be suitable and vary in quality. For distinguishing and validating the solution domain knowledge sources we basically consider the quality factors of *objectivity* and *relevancy*. The objectivity quality factor refers to the solution domain knowledge sources itself, and defines the general acceptance of the knowledge source. Solution domain knowledge that is based on a consensus on a community of experts has a higher objectivity degree than solution domain knowledge that is just under development. The relevancy factor refers to the relevancy of the solution domain knowledge for solving the identified technical problem.

The relevancy of the solution domain knowledge is different from the objectivity quality. A solution domain knowledge entity may have a high degree of objective quality because it is very precisely defined and supported by a community of experts, though, it may not be relevant for solving the identified problem because it addresses different concerns. To be suitable for solving a problem it is required that the solution domain knowledge is both objective and relevant. Therefore, the identified solution domain knowledge is prioritized according to their objectivity and relevancy factors. This can be expressed in the empirical formula [Aksit 00]:

$$priority(s) = (objectivity(s), (relevance(s))$$

Hereby *priority*, *objectivity* and *relevance* represent functions that define the corresponding quality factors of the argument *s*, that stands for solution domain knowledge source. For solving the problem, first the solution domain knowledge with the higher priorities is utilized. The measure of the objectivity degree can be determined from general knowledge and experiences. The measure for the relevancy factor can be determined by considering whether the identified solution domain source matches the goal of the problem. Note, however, that this formula should not be interpreted too strictly and rather be considered as an intuitive and practical aid for prioritizing the identified solution domain knowledge sources rather.

Example

Let us now consider the identification and the prioritization of the solution domains for the given project example. For the overall problem a solution is provided by the solution domain *Atomic Transactions*. Table 1 provides the solution domains for every sub-problem.

SUB-PROBLEM	SOLUTION DOMAIN
P1	Transaction Management
P2	Concurrency Control
P3	Recovery
P4	Adaptability

Table 1. The solution domains for the sub-problems

The prioritization of these solution domains was defined as in the above order from the top to the bottom.

For the overall problem and the corresponding solution domain of *Atomic Transactions*, we could find sufficient knowledge sources. Our identified solution domain knowledge sources consisted of managers, system developers, maintainers, documentation, literature on transactions and the existing NEDIS system. However, among these different knowledge sources we assigned higher priority values to the literature on atomic transaction systems. Table 2 provides the selected set of knowledge sources for the overall solution domain.

ID	KNOWLEDGE SOURCE	FORM
KS1	*Concurrency Control & Recovery in Database Systems [Bernstein et al. 87]*	Textbook
KS2	*Atomic Transactions [Lynch et al. 94]*	Textbook
KS3	*An Introduction to Database Systems [Date 90]*	Textbook
KS4	*Database Transaction Models for Advanced Applications [Elmagarmid 92]*	Textbook
KS5	*The design and implementation of a distributed transaction system based on atomic data types [Wu et al. 95]*	Journal. paper
KS6	*Transaction processing: concepts and techniques [Gray & Reuter 93]*	Textbook
KS7	*Principles of Transaction Processing [Bernstein & Newcomer 97]*	Textbook
KS8	*Transactions and Consistency in Distributed Database Systems [Traiger et al. 82]*	Journal paper

Table 2. A selected set of the identified knowledge sources for the overall solution domain

The table consists of three columns that are labeled as *ID*, *Knowledge Source* and *Form* that respectively represent the unique identifications of the knowledge sources, the title of the knowledge source and the representation format of the knowledge source. The table includes the knowledge sources that describe atomic transactions in a general way. Knowledge sources that deal with a specific aspect of transaction systems, for example such as deadlock detection mechanisms, have been temporarily omitted and are identified when the corresponding sub-problems are considered.

In the same manner we looked for knowledge sources for the individual sub-problems and we were able to identify many knowledge sources for the solution domains *Transaction Management*, *Concurrency Control* and *Recovery*. The solution domain *Adaptability* was more difficult to grasp than the other ones. For this we did a thorough analysis on the notion of adaptability and studied various possibly related publications such as control theory [Roxin 97][Foerster 79][Umplebey 90]. In addition we organized a workshop on Adaptability in Object-Oriented Software Development [Tekinerdogan & Aksit 97] [Aksit et al. 96].

As an example, Table 3 shows a selected set of the identified knowledge sources for the solution domain *Concurrency Control*.

ID	KNOWLEDGE SOURCE	FORM
KS1	*Concurrency Control in Advanced Database Applications [Barghouti & Kaiser 91]*	Journal paper
KS2	*Concurrency Control in Distributed Database Systems [Cellary et al. 89]*	Textbook
KS3	*The theory of Database Concurrency Control [Papadimitriou 86].*	Textbook
KS4	*Concurrency Control & Recovery in Database Systems [Bernstein et al. 87]*	Textbook
KS5	*Concurrency Control and Reliability in Distributed Systems [Bhargava 87]*	Journal paper
KS6	*Concurrency Control in Distributed Database Systems [Bernstein & Goodman 83]*	Textbook

Table 3. A selected set of the identified knowledge sources for the solution domain CONCURRENCY CONTROl

Note that the knowledge source KS4 has also been utilized for the overall solution domain. The reason for this is that this knowledge source is both sufficiently abstract to be suitable for the overall solution domain and provides detailed information on the solution domain *Concurrency Control*.

Extract Solution Domain Concepts from Solution Domain Knowledge

Once the solution domains have been identified and prioritized, the knowledge acquisition from the solution domain sources can be initiated. The solution domain knowledge may include a lot of knowledge that is covered by books, research papers, case studies, reference manuals, existing prototypes/systems etc. Due to the large size of the solution domain knowledge, the knowledge acquisition process can be a labor-intensive activity and as such a systematic approach for knowledge acquisition is required [Partridge & Hussain 95], [Gonzales & Dankel 93], [Wielinga et al. 92].

In our approach we basically distinguish between the *knowledge elicitation* and *concept formation* process. Knowledge elicitation focuses on extracting the knowledge and verifying the correctness and consistency of the extracted data. Hereby, the irrelevant data is disregarded and the relevant data is provided as input for the concept formation process. Knowledge elicitation techniques have been described in several publications and its role in the knowledge acquisition process is reasonably well-understood [Wielinga et al. 92], [Meyer & Booker 91], [Diaper 89a], [Firlej & Hellens 91].

The concept formation process utilizes and abstracts from the knowledge to form concepts[5]. In the literature, several concept formation techniques have been identified[6] [Parsons & Wand 97][Reich & Fenves 91][Lakoff 87]. One of the basic abstraction techniques in forming concepts is by identifying the variations and commonalities of extracted information from the knowledge sources [Stillings et al. 95][Howard 87]. Usually a concept is defined as a representation that describes the common properties of a set of instances and is identified through its name.

Example

We analyzed and studied the identified solution domain knowledge according to the defined priorities and extracted the fundamental concepts. After considering the commonalities and variabilities of the extracted information from the solution domains we could extract the following solution domain concepts:

ATOMIC TRANSACTION SYSTEMS

An atomic transaction system is a well-known and fundamental abstraction which provide the necessary concurrency control and recovery mechanisms for the application programs. Transactions relieve application programmers of the burden of considering the effects of concurrent access to objects or various kinds of failures during execution. Transactions simplify the treatment of failures and concurrency and may thereby provide the application programmer location transparency, replication transparency, concurrency transparency and failure transparency. Informally atomic transactions are characterized by two properties: *serializability* and *recoverability* [Bernstein et al. 87]. Serializability means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same set of

[5] Recall from chapter 3 that there are basically three views of concepts, including the classical view, the prototype view and the exemplar view. Concept forming through abstraction from instances is basically applied in the classical view and the prototype view [Lakoff 87].

[6] This process of concept abstraction is usually considered as a psychological activity that is often associated with the term 'experience' [Stillings et al. 95]. Experts, i.e. persons with lots of experience, own a larger set of concepts and are better in forming concepts than persons who lack this experience.

transactions. Recoverability means that each execution appears to be all or nothing; either it executes successfully to completion or it has no effect on data shared with other transactions.

TRANSACTION

The concept *Transaction* represents a transaction block as defined by the programmer.

TRANSACTION MANAGEMENT

The concept *TransactionManager* provides mechanisms for initiating, starting and terminating the transaction. It keeps a list of the objects that are affected by the transaction. If a transaction reaches its final state successfully, then *TransactionManager* sends a commit message to the corresponding objects to terminate the transaction. Otherwise an abort message is sent to all the participating objects to undo the effects of the transaction. The *TransactionManager* concept includes knowledge about a variety of commit and abort protocols.

POLICY MANAGEMENT

The concept *PolicyManager* determines the mechanisms for adapting transaction protocols. In most publications, the *PolicyManager* is included in the *TransactionManager*. We considered defining transaction policies as a different concern and therefore defined it as a separate concept.

SCHEDULER

The concept *Scheduler* is responsible for the concurrency control mechanism. It provides the concurrency control by restricting the order in which the operations are processed. Incoming operations may be accepted, rejected or put in a delay queue. Concurrency control may be based on syntactic ordering of the operations (e.g. read, write) or it may use semantic information of the transaction, such as information on the accessed data types. Traditional concurrency control techniques are locking, timestamp ordering and optimistic scheduling.

RECOVERY MANAGER

The concept *Recovery Manager* is responsible for the recovery in case of transaction aborts, system failures and/or media failures. Failures may have an effect on data objects and on transactions that read the data objects. Recovery of the data objects needs caching and undo/redo mechanisms. Recovery of the effected transactions requires scheduling for recovery so that failures are prevented.

DATA MANAGER

The concept *DataManager* controls the access to its object and keeps it consistent by applying concurrency control and recovery mechanisms. Further it may be responsible for the version management and the replication management of the data objects.

DATA OBJECT

The concept *Data Object* represents a data object that needs to be accessed in a consistent way. This means that the object must fulfill the consistency constraints set by the application.

Structure the Solution Domain Concepts

The identified solution domain concepts are structured into *taxonomies* and *partimonies* using *specialization*- and *aggregation* relations respectively. In addition, also other structural *association* relations are used. Like the concepts themselves the structural relations between the concepts are also derived from the solution domains.

For the structuring and representation of concepts so-called *concept graphs* are used. A *concept graph* is a graph which nodes are consisting of *concepts* and the edges between the nodes represent *conceptual relations*. The notation of concept graphs is given in the following figure:

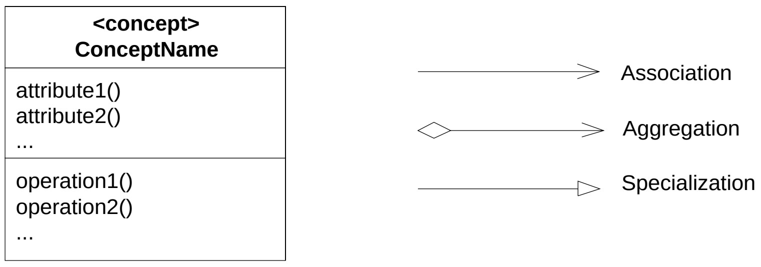

Figure 10. Notation for Concept Graphs

The notation for a concept is a stereotype of the class notation in the Unified Modeling Language [Booch et al. 99]. A stereotype represents a subclass of a modeling element with the same form but with a different intent. The stereotype for a concept Figure 10 is identified by the keyword <concept>[7].

Example

Figure 11 shows the structuring of the solution domain concepts in the top-level concept graph of transaction systems. The concept *Transaction Manager* has an association relation *manages* with the concept *Transaction Application*. This means that *Transaction Manager* is

[7] Note that a class may not be similar to a concept. Although both classes and concepts are generally formed through an abstraction process this does not imply that every abstraction is a concept. A concept is a well-defined and stable abstraction in a given domain.

responsible for the atomic execution of *TransactionApplication*. The association relation *manages* between concept *DataManager* and *Atomic Object* defines the consistency maintenance.

For keeping the *Atomic Object* consistent the *Datamanager* utilizes and coordinates the concepts *Scheduler* and *Recovery Manager* by means of the association relation *coordinates*. The concept *PolicyManager* coordinates the activities of the concepts *Transaction Manager* and *Data Manager* and defines the adaptation policy. Finally, the association relation *accesses* between *Transaction Application* and *Atomic Object* defines a read / update relation between these two.

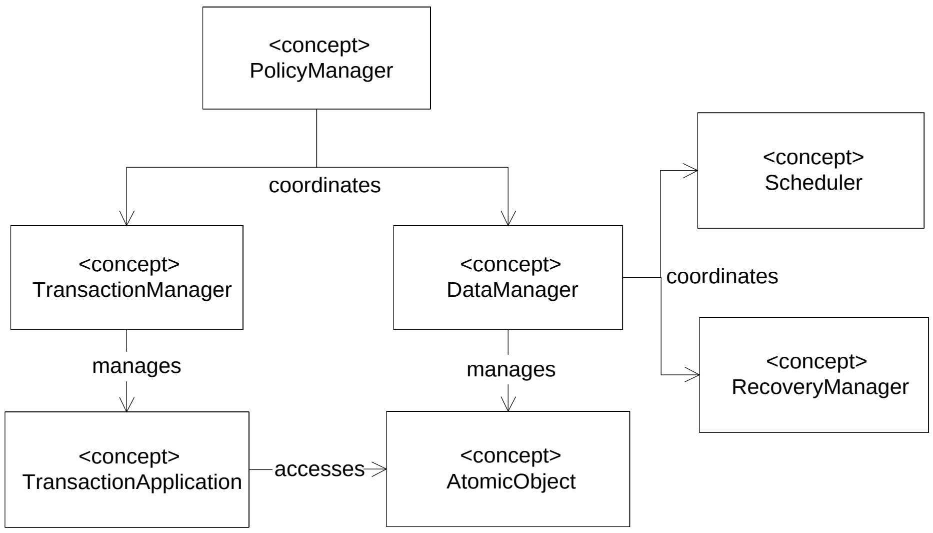

Figure 11. The top-level concept graph of an atomic transaction system

Refinement of Solution Domain Concepts

After identifying the top-level conceptual architecture we focus on each sub-problem and follow the same process. Recall that in Figure 5, this refinement process is represented by the arrow directed from plan 0.3 to plan 0.1. The refinement may be necessary if the architectural concepts have a complex structure themselves and this structure is of importance for the eventual system.

The ordering of the refinement process is determined by the ordering of the problems with respect to their previously determined priorities. Architectural concepts that represent problems with higher priorities are handled first. In the following we will refine the architectural concepts according to this ordering. The refinement requires executing the plans 0.1 to 0.3 for each selected concept. However, due to space limitations we will only describe these plans globally.

Example

In the following we will shortly describe the refinement for each concept of the atomic transaction architecture.

Refining the TransactionManager concept

To refine the TransactionManager concept we looked for the knowledge sources that specifically dealt with transaction management or included detailed information about this. We identified several publications for this purpose [Elmagarmid 92][Bernstein & Newcomer 97],[Moss 85][Jajodia & Kerschberg 97].

In parallel with the solution domain analysis process we tried to refine problem P3 for transparent transaction management, as it has been described in the problem structure diagram in Figure 7. This resulted in the definition of the sub-problems *P3.1 Start Protocol*, describing the need for defining a transaction start protocol, *P3.2 Commit/Abort Protocol*, describing the need for a commit / abort protocol and *P3.3 Nested Transactions*, describing the need for nested transactions. The specifications of these sub-problems were again defined in close interaction with the client. After comparison of the concepts in these knowledge sources we could extract the commonalities and derive the architecture for the concept TransactionManager as it is given in Figure 12.

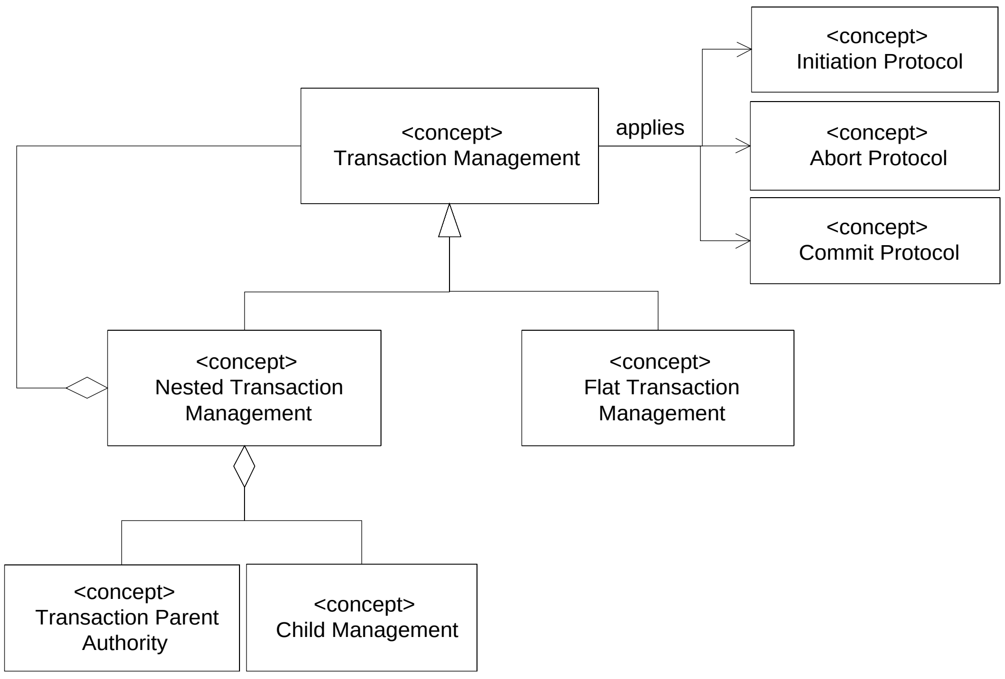
<concept> Initiation Protocol
<concept> Transaction Management
applies
<concept> Abort Protocol
<concept> Commit Protocol
<concept> Nested Transaction Management
<concept> Flat Transaction Management
<concept> Transaction Parent Authority
<concept> Child Management

Figure 12. Conceptual Architecture of TransactionManager

The concept *Transaction Manager* applies the concepts *Initiation Protocol*, *Abort Protocol* and *Commit Protocol* for starting and terminating a transaction. The *Initiation Protocol* represents the starting of the transaction and prepares the program to be executed. The *Abort Protocol* and *Commit Protocol* concepts refer to the protocols that terminate the transaction. The *Abort Protocol* will be executed if the transaction has failed and its effects on the data objects and the other transactions need to be restored. The *Commit Protocol* will be executed if the transaction protocol has succeeded and the results need to be made persistent.

Transactions may consist of other transactions as well. The composition of sub-transactions into one transaction is called a nested transaction [Moss 85]. Hereby the transactions are hierarchically ordered whereby a *parent transaction* includes several other sub-transactions. The advantages of nested transactions over flat transactions is that they provide internal parallelism of the sub-transactions and finer control over failures by limiting the effects of the failure to a sub-transaction. This is especially important for long and complex transactions that have, for example, higher failure risks. In Figure 12, the concept *Transaction Parent Authority* refers to the authority of the parent on the sub-transactions with respect to the commit protocols. In the literature, basically a distinction is made between *closed nested transactions* and *open nested transactions* [Elmagarmid 92]. In *closed nested transactions* the sub-transactions are not allowed to commit before the parent transaction commits, whereas in *open nested transactions* the sub-transactions can commit before the parent commits. The concept *Child Management* refers to the composition strategy of the sub-transactions into a one complete transactions. Usually this is done at compile time, but several approaches have illustrated the practical use of dynamic composition and decomposition of sub-transactions [Pu et al. 88]

Refining the DataManager concept

Figure 13 shows the architecture for the concept *DataManager*. The basic knowledge sources that we adopted to identify the common abstractions of data management techniques are derived from several publications [Weihl 90][Wu et al. 95][Guerraoui 94].
The concept *Datamanager* coordinates the concepts *Scheduler* and *RecoveryManager,* which are respectively responsible for the scheduling of the incoming concurrent operations and the recovery in case of failures. In addition the concept *DataManager* uses the concepts *VersionManager* and *ReplicationManager* for respectively managing multiple versions of the data item and the replication of it at different locations. The version management and the replication management were not addressed as separate problems in the problem analysis phase. After interaction with the client it was decided to omit these two issues and only consider the concurrency control and recovery in the data management. If these were

addressed as important problems then we would update the problem structure diagram and attempt to provide solutions for these problems in the later phases of the approach.

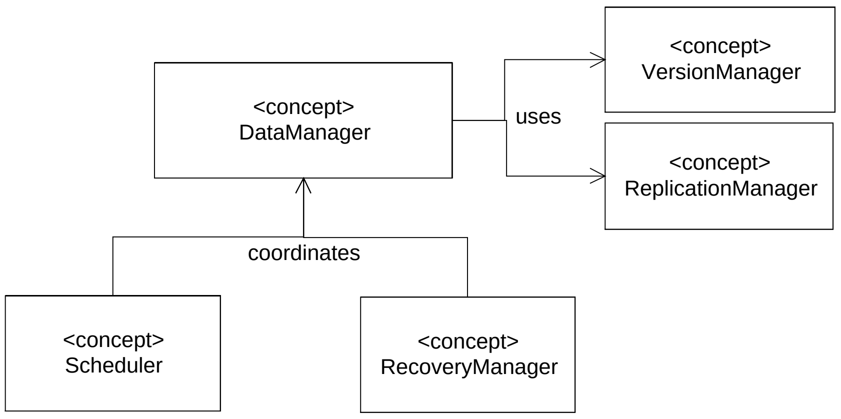

Figure 13. Conceptual Architecture of DataManager

Refining the Scheduler concept

Figure 14 represents the architecture of the concept *Scheduler*. The selected knowledge sources that we identified to extract this structure have been listed in Table 3 in the sub-section on identifying and prioritizing knowledge sources. In parallel with the solution domain analysis we refined the problem structure diagram for the concept *Scheduler* and added the sub-problems *P1.1 Syntactic Synchronization* and *P1.2 Performance Failure Detection*. These problems correspond to the solution domain concepts in the conceptual architecture of *Scheduler* that consists of three sub-concepts: *Synchronization Scheme, Synchronization Strategy* and *Performance Failure Detector.*

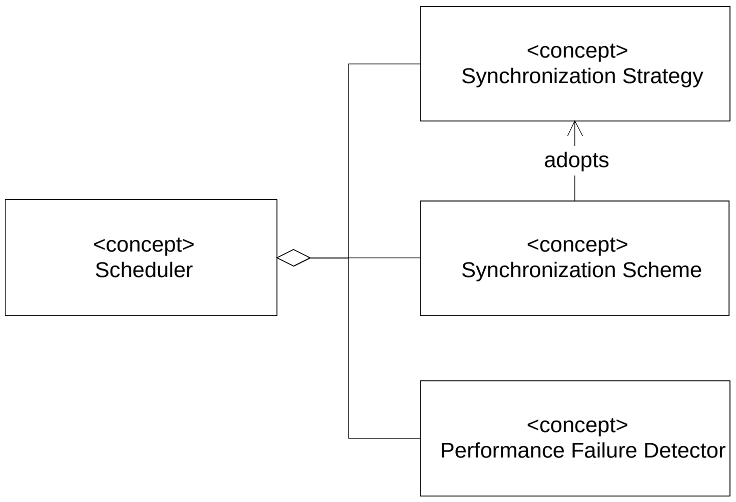

Figure 14. Conceptual Architecture of Scheduler

The concept *Synchronization Scheme* defines the synchronization approach by accepting, rejecting or delaying the incoming operations. It addresses the problem *P1.1 Syntactic Synchronization* in the problem structure diagram of Figure 7. The syntactic synchronization may be basically through locking, timestamp-ordering and optimistic concurrency control schemes. The concept *Synchronization Strategy* also addresses the problem *P1.1 Syntactic Synchronization* and refers to the adopted strategy in the applied concurrency control algorithm. Basically a distinction is made between conservative and aggressive schedulers. A conservative scheduler tends to delay operations whereas an aggressive scheduler avoids these delays and aborts the operation sooner. The concept *Performance Failure Detector* addresses the problem *P1.2 Performance Failure Detection* and concerns the detection of performance failures such as deadlocks that are side effects of the used concurrency control algorithms.

Refining the Recovery Manager Concept

The concept *RecoveyManager* is related to the problem *P2. Transparent Recovery* that is depicted in the problem structure diagram in Figure 7. It has been derived from the publications on recovery in transaction systems [Bernstein et al. 87][Bhargava et al. 86][Hadzilacos 88][Haerder & Reuter 83]. In parallel with refining the concept *RecoveyManager* we refined problem P2 and defined the sub-problems *P2.1 Recovery from Transaction Failures* and *P2.2 Recovery from System Failures*. The architecture for the concept *RecoveryManager* is given in Figure 15.

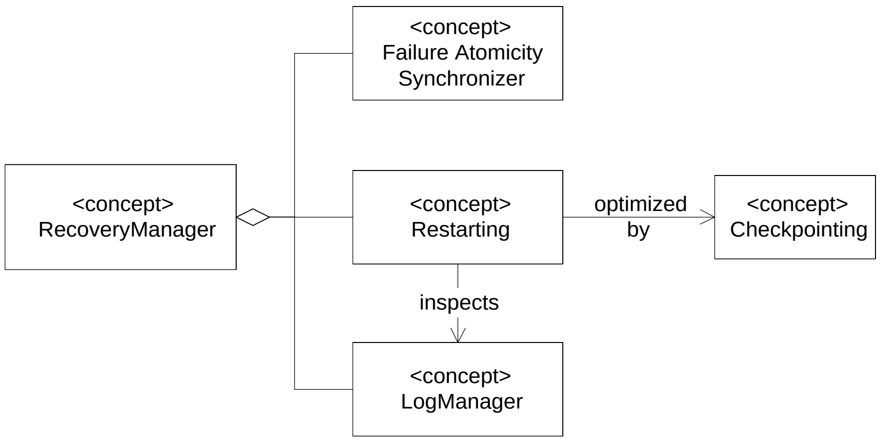

Figure 15. Conceptual Architecture of RecoveryManager

The concept *RecoveryManager* consists of four sub-concepts *Failure Atomicity Synchronizer, Restarting, LogManager* and *Checkpointing*. The effects of a transaction can be both on the accessed data objects and on other transactions that access the same data object. To undo the effects of failures on data objects the sub-concept *LogManager* is used for logging the data object. The sub-concept *Failure Atomicity Synchronizer* order transaction operations to provide the all-or-nothing property. The sub-concept *Restarting* is responsible for recovering from system failures and initializes the transaction to its last recoverable state by inspecting the logs of the *LogManager*. Thereby it uses algorithms for undoing the actions of aborted and active transactions and redoing the effects of transactions that have been committed before but not made persistent yet. The sub-concept *Checkpointing* represents the optimization of the restart process by making a snapshot or checkpoint of the basic events in the system that may be used by the protocols of the concept *Restarting*.

Refining the Policy Manager Concept

The concept *PolicyManager* is related to the technical problem *P4. Provide Adaptable Transaction Protocols*. The identified knowledge sources for the concept *PolicyManager* have been derived from several publications on control systems [Dorf & Bishop 98][Shinners 98] and performance modeling [Kumar 96][Atkins & Coady 92][Highleyman 89][Agrawal 87][Carey 84]. The *PolicyManager* evaluates a number of performance metrics and selects the preferred transaction protocols with respect to these parameters. Examples of performance metrics are the following:

1. *Transaction throughput rate*, which is the number of transactions completed per second.
2. *Response time*, which is the measure of the time difference between a transaction initiation and a successful termination of the transaction.
3. *Blocking ratio,* which is the average number that a transaction has to block per commit.
4. *Restart ratio,* which is the average number that a transaction has to restart per commit.

The conceptual architecture of *PolicyManager* is given in Figure 16. It consists of the sub-concepts *Sensor*, *Comparator*, *Decider* and *Actuator*.

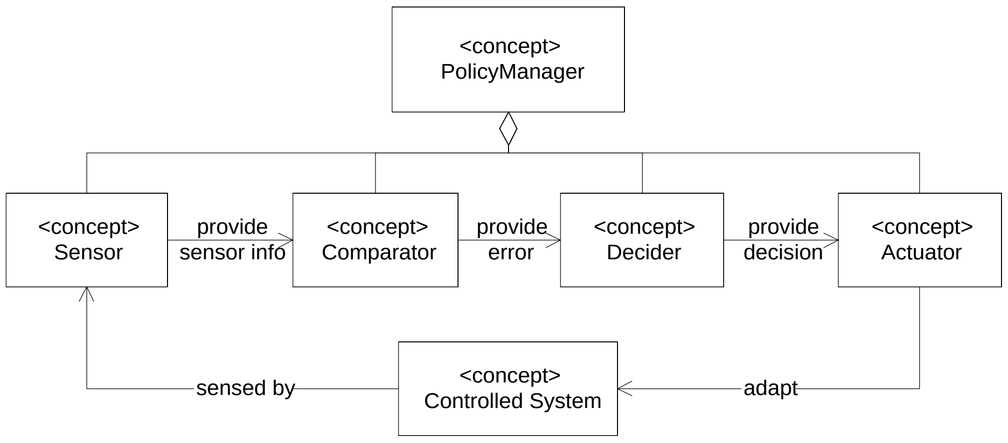

Figure 16. Conceptual Architecture of PolicyManager

The concept *Sensor* keeps track of the changes in the system state and the values of the performance parameters and provides this information to the sub-concept *Comparator* that compares this information according to the initialized criteria and goals that need to be met. For example, *Comparator* may have defined different threshold values for the *throughput* parameter and compares this with the perceived values of the *throughput* parameter and inform the sub-concept *Decider* about the difference. *Decider* will then select an adequate transaction protocol and inform the sub-concept *Actuator* about this decision. *Actuator* will actually adapt the system with the required transaction protocols. Note that this architecture may be implemented in various ways such as a very adaptation algorithm or an expert-system based selection.

4.4.4 Alternative Space Analysis

We define the *alternative space* as the set of possible design solutions that can be derived from a given conceptual software architecture. The *Alternative Design Space analysis* aims to depict this space and consists of the sub-processes *Define the Alternatives for each Concept* and *Describe the Constraints*. Let us now explain these sub-processes in more detail.

Define the Alternatives for each Concept

In the synthesis-based design approach the various architecture design alternatives are largely dealt with by deriving architectural abstractions from well-established concepts in the solution domain that have been leveraged to the identified technical problems. Each architectural concept is an abstraction from a set of instantiations and during the analysis and design phases the architecture is realized by selecting particular instances of the architectural concepts. An instance of a concept is considered as an alternative of that concept. The total set of alternatives per concept may be too large and/or not relevant for solving the identified problems. Therefore, to define the boundaries of the architecture it is necessary to identify the relevant alternatives and omit the irrelevant ones.

Let us now consider the process of alternative selection. The alternatives of a given concept may be explicitly identified and published [Tekinerdogan 94]. In that case, selecting alternatives for a concept is rather straightforward and depends only on the solution domain analysis process. If the concepts have complex structures consisting of sub-concepts then an alternative is defined as a composition of instances of separate sub-concepts. The set of alternatives may be too large to provide a name for each of them individually. Nevertheless, we need to depict the total set of alternatives so that every one of them can be derived if necessary. We do this by identifying the alternatives of each sub-concept first and then considering the various compositions of these alternatives to provide the higher-level alternatives.

Example

Let us now consider the alternatives for the concepts in the top-level architecture. We depict the alternative space by providing a table in which the column headers represent the sub-concepts and each table entry represents an instance of the sub-concept in the column header. For example, Table 4 represents the alternative space for the concept *Scheduler.* It has 4 columns, the first one represents the numbering of alternatives and the second to the fourth columns represents the sub-concepts of the concept *Scheduler.*

	A. SYNCHRONIZATION SCHEME	**B. SYNCHRONIZATION STRATEGY**	**C. PERFORMANCE FAILURE DETECTOR**
1.	Two Phase Locking	Aggressive	Deadlock Detector
2.	Timestamp Ordering	Conservative	Infinite Blocking Detector
3.	Optimistic		Infinite Restart Detector
4.	Serial		Cyclic Restart Detector

Table 4. Alternatives of the sub-concepts of *Scheduler*

The sub-concept *Synchronization Decision* has four alternatives, namely, *Two-phase locking, Timestamp Ordering, Optimistic* and *Serial.* The sub-concept *Synchronization Strategy* has the alternatives *Aggressive* or *Conservative.* The alternatives of the sub-concept *Performance Failure Detector* detect *performance failures* that may consist of *deadlock, permanent blocking, cyclic restarting* and *infinite restarting. Deadlock* is defined as a state where two transactions are mutually waiting for each other to release data objects necessary for their completion. *Permanent blocking* occurs when a transaction waits indefinitely for a data object granting because of a steady stream of other transactions whose data access requests are always granted before. *Cyclic restarting* occurs when two or more transactions continually cause mutual abortion of each other. *Infinite restarting* occurs when a transaction is infinitely aborted because of a steady stream of other transactions whose operations are always granted before.

An alternative of the concept *Scheduler* is a composition of selections of the alternatives of the sub-concepts. For instance, an alternative that may be derived from Table 4 is the tuple *(Two Phase Locking, Conservative, Deadlock Detector*) which represents a scheduler that uses aggressive two phase locking protocol whereby a deadlock detection mechanism is used to remove the deadlocks that may occur.

Table 5 represents the alternative space for the concept *RecoveryManager.*

	A. LOGMANAGER	B. FAILURE ATOMICITY SYNCHRONIZER	C. RESTARTING	D. CHECKPOINTING
1.	Operation Logging	Recoverable	Undo / Redo	Commit-Consistent
2.	Deferred-Update	Cascadeless	No-Undo / Redo	Cache-Consistent
3.	Update-In-Place	Strict	Undo / No-Redo	Fuzzy
			No-undo / No-redo	

Table 5. Alternatives of the sub-concepts of RecoveryManager

The sub-concept *LogManager* consists of three alternatives of logging techniques. *Operation Logging* the transaction's operations that access a data object are logged. In case of aborts other operations are executed to undo the effects of the operations that were logged. Another logging technique is to make a copy of the state of the object, which is called *image logging*. Hereby, either the copy may be accessed or the original. The former is called *Deferred-Update Logging* because updates to the original data objects are deferred until commit time. The latter is called *Update-In-Place* logging whereby the copy of the data object is installed on abort and originals are left on commit. The sub-concept *Failure Atomicity Synchronizer* orders operation in three possible ways and provides either *recoverable*, *cascadeless aborts* or *strict* executions. Restarting can be performed as a combination of undo and redo protocols and as such there are four alternatives here. Finally, the sub-concept *Checkpointing* consists of the three alternatives *Commit-Consistent*, *Cache-Consistent* and *Fuzzy* checkpointing mechanisms.

An alternative of the concept *RecoveryManager* is the tuple *(Operation Logging, Strict, Undo-Redo, Commit-consistent)*, that represents a *RecoveryManager* which applies *Operation Logging*, *Strict* executions, adopts *Undo-Redo* algorithm in case of restarts and a *Commit-Consistent* checkpointing mechanism for optimizing the restart procedure.

Describe Constraints between Alternatives

An architecture consists of a set of concepts that are combined in a structure. An instantiation of an architecture is a composition of instantiations of concepts [Aksit et al. 99][Aksit et al. 98]. The instantiations of these various concepts may be combined in many different ways and likewise this may lead to a combinatorial explosion of possible solutions. Hereby, it is generally impossible to find an optimal solution under arbitrary constraints for an arbitrary set of concepts.

To manage the architecture design process and define the boundaries of the architecture it is important to adequately leverage the alternative space. Leveraging the alternative space means the reduction of the total alternative space to the relevant *alternative space*. A reduction in the space is defined by the solution domain itself that defines the *constraints* and as such the possible combination of alternatives. The possible alternative space can be further reduced by considering only the combinations of the instantiations that are relevant from the client's perspective and the problem perspective.

Constraints may be defined for the sub-concepts within a concept as well as among higher-level concepts. We describe first the constraints among the sub-concepts within a concept and later among the concepts. *Binary constraints* are the constraints among two concepts. Constraints may be also defined for more than two concepts together. We use the *Object Constraint Language* (OCL) [Warmer & Kleppe 99] that is part of the UML to express the constraints over the various concepts.

Constraint identification is not only useful for reducing the alternative space but it may also help in defining the right architectural decomposition. The existence of many constraints between the architectural components provides a strong coupling and as such it may refer to a wrong decomposition. This may result in a reconsideration of the identified architectural structure of each concept.

Example

From the solution domain we could identify several constraints that restrict the alternative space of the architecture. In the following we will describe examples of the constraints for the sub-concepts of the concepts *Scheduler* and *RecoveryManager* [Weihl 90][Weihl 89][Guerraoui 94] using the Object Constraint Language (OCL). In addition we will provide the constraints among the concepts *Scheduler* and *RecoveryManager*.

Figure 17 illustrates three constraints for the sub-concepts of *Scheduler*. The first constraint defines that for a scheduler with a *two-phase locking* synchronization scheme and a synchronization strategy that is *conservative* either a *deadlock detector* or an *infinite blocking detector* is needed. The reason for this is that the other two performance failures, *infinite restart* and *cyclic restart*, can never occur for this alternative of a scheduler [Cellary et al. 89]. The second constraint indicates that optimistic and timestamp ordering schedulers either need an infinite restart or cyclic restart detector. Finally, the third constraint defines that a serial scheduler does not lead to performance failures because it orders operations of transactions serially and never delays or aborts transactions.

1. Conservative Two Phase Locking schedulers need only either a Deadlock Detector or an Infinite Blocking Detector.

```
if  (self.SynchronizationScheme = 'TwoPhaseLocking')
        and (self.SynchronizationStrategy='Conservative')
then (self.PerformanceFailureDetector='Deadlock Detector' ) or
        (self.PerformanceFailureDetector='Infinite Blocking Detector')
endif
```

2. Optimistic and timestamp ordering schedulers need only detectors for either an infinite restart or a cyclic restart.

```
if  (self.SynchronizationScheme = 'Optimistic')  or
        (self.SynchronizationScheme = 'Timestamp Ordering')
then (self.PerformanceDailureDetector='Infinite Restart Detector' ) or
        (self.PerformanceFailureDetector='Cyclic Restart Detector')
endif
```

3. A serial scheduler does not need to detect failures.

```
if  (self.SynchronizationScheme = 'Serial')
then self.PerformanceFailureDetector= nil
endif
```

Figure 17. Constraints for the sub-concepts of *Scheduler*

Figure 18 illustrates the constraints for *RecoveryManager*. The first constraint defines that a deferred-update recovery technique does not require an undo process in case of restarts. This is because original data objects are not accessed during the execution of transactions and only the copies are affected. The second constraint defines that an update-in-place does not require a redo process. The reason for this is that the original data objects are already accessed during execution of transactions and on commit it is not necessary anymore to install the effects of the transactions.

Figure 19 illustrates a constraint among the concept *Scheduler* and *RecoveryManager*. It defines that a serial scheduler will not use a synchronization protocol to provide atomicity, simply because no concurrency is allowed for this scheduler.

```
1.  Deferred-Update does not require undo process
if  (self.LogManager = 'Deferred-Update')
then (self.Restarting = 'No-Undo/Redo' or self.Restarting = 'No-Undo/No-Redo')
endif

2.  Update-In-Place does not require redo process
if  (self.LogManager = 'Update-In-Place Logging')
then (self.Restarting = 'No-Redo/Undo' or self.Restarting 'No-Redo/No-Undo')
endif
```

Figure 18. Constraints for the sub-concepts of *RecoveryManager*

```
1.  A serial scheduler does not synchronize operations for recovery.
if  (scheduler.SynchronizationScheme = 'Serial')
then (recoveryManager.FailureAtomicitySynchronizer=nil)
endif
```

Figure 19. A constraint between *Scheduler* and *RecoveryManager*

Other constraints are identified for example for the commit and abort protocols of the *TransactionManager,* which must be understood by the different data managers in the system. If the protocols of the *TransactionManager* are changed, then the protocols of the data managers must change accordingly. Due to space limitations we will not further elaborate on the other constraints in this thesis.

4.4.5 Architecture Specification

It consists of the two sub-processes *defining semantics of the architecture* and *defining dynamic behavior of the architecture.*

Defining Semantics of the Architecture

We consider each concept separately to derive its semantics from the solution domains to provide a more formal, but corresponding, specification. As a format for writing a formal specification we use:

<operation><pre-condition><post-condition>

Hereby, <operation> represents the name of the operation of a concept. The name and the type of each concept variable are described in the part <declarations>. The part <pre-conditions> describe the conditions and assumptions made about the values of the concept variables at the beginning of

<operation>. The part <post-conditions> describe what should be true about the values of the variables upon termination of <operation>.

Example

Creating and Terminating Transactions

The architecture component *Transaction* represents the application program that is executed as an atomic transaction. We can derive the semantics for this component from the solution domain. For example, the following represents the semantics of *Transaction,* as it has been adapted from [Lynch et al. 94].

Transaction::Start
postcondition:
self.status="running"

Transaction::Commit
postcondition:
self.status="success"

Transaction::Abort
postcondition:
self.status="fail"

….
// *Additional operations*

Figure 20. Specification of the interface of Transaction

A transaction can be started using the operation *Start,* which initializes the transaction parameters and may include application specific operations before the starting the transaction. The variable *status* represents the state of the transaction and can have the values *running, success or fail.* The operations *Commit* and *Abort* respectively commit and abort the transaction and may include specific operations after the termination of the transaction. These three operations are generic for most transaction applications. Other operations may be added for specific transaction applications. For example, in a car dealer system, operations such as *Reserve_Car*, *Order_Car* and *Request_CarInfo* would be defined.

Every transaction will be managed by a *TransactionManager* that is basically responsible for the creation, initialization and termination of the transactions. The semantics of *TransactionManager* for managing flat transactions as adapted from [Bernstein et al. 87] is presented in Figure 21:

```
TransactionManager::Start(T:Transaction)
postcondition:
        self.transaction_started=true;
        self.transaction = T;

TransactionManager::Request_Commit(T:Transaction)
precondition:
        self.transaction_started= true;
        self.transaction_committed = false;
        self.transaction_aborted=false;
postcondition:
        self.transaction_commitrequested= true;

TransactionManager::Commit(T:Transaction)
precondition:
        self.transaction_commitrequested= true;
postcondition:
        self.transaction_committed= true;
        self.transacttion = nil.

TransactionManager::Abort(T:Transaction)
precondition:
        self.transaction_started = true;
        self.transaction_committed = false;
        self.transaction_aborted=false;
postcondition:
        self.transaction_aborted=true;
        self.transaction = nil;
```

Figure 21. Specification of the interface for TransactionManager dealing with flat transactions

TransactionManager includes the operations *Start*, *RequestCommit*, *Commit* and *Abort*. Further it includes 5 boolean variables *transaction_started*, *transaction_committed*, *transaction_aborted*, *commit_requested* and one variable *transaction* that keeps the *Transaction* object. The operation *Start* initates the transaction and sets the boolean variable *transaction_started* to true. The initiation of a transaction may include the assignment of, for example, unique transaction id and/or timestamp. A commit of a transaction must always be requested before, which is done by executing the operation *Request_Commit*. This operation is only allowed if the transaction has been started but not completed yet. The operations *Commit* and *Abort* terminate the transaction, set the boolean variables accordingly and initialize the variable *transaction* to nil, so that a new transaction can be started.

TransactionManager component may also express nested transactions. The specification of additional operations for a transaction manager for nested transactions that we have adapted from the solution domain on nested transaction [Moss 85] is given in Figure 22.

TransactionManager::Create_SubTransaction(T:Transaction)
postcondition:
self.subtransactions = self.subtransactions ∪{T};
T.parent = self.

TransactionManager::getParent()
postcondition:
if self.transaction is not top_transaction then *return* self.parent
else return self

TransactionManager::Report_Commit(T:Transaction)
postcondition:
parent.committed_subtransactions = parent.committed_subtransactions ∪{T};

TransactionManager::Report_Abort(T:Transaction)
postcondition:
parent.aborted_subtransactions = parent.aborted_subtransactions ∪{T};

Figure 22. Specification for additional operations of TransactionManager for nested transactions

The operation *Create_Subtransaction* creates a subtransaction for the corresponding transaction and sets the *parent* of the sub-transaction. A parent transaction is not allowed to complete its own activity until its subtransactions have terminated. This means that the commit and abort operations need to be implemented accordingly. If a subtransaction aborts, the parent can choose different actions, such as ignoring, triggering another transaction or aborting itself. In flat transactions, after the confirmation of a commit from the datamanagers the transaction was able to commit or to abort. In nested transactions, the transaction needs first to report the result of the termination to its parent transaction. In Figure 22, the operations *Report_Commit* and *Report_Abort* inform the parent transactions on respectively the commit and the abort of the sub-transaction. Depending on whether open nested transactions or closed nested transactions are implemented the implementation of the commit and abort operations may change accordingly.

Scheduler

The component *Scheduler* deals with the concurrency control of transaction operations in order to keep the data object consistent. Figure 23 represents an example of the semantics of the Scheduler that is based on two phase locking.

```
Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "read"
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m);
postcondition:
    self.readlock_holders = self.readlock_holders ∪ (T,m);

Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "write"
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m);
postcondition:
    self.writelock_holders = self.writelock_holders ∪ (T,m);

Scheduler::CommitRequest(T:Transaction)
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m);
postcondition:
    self.commit_requested = self.commit_requested ∪ T;

Scheduler::Commit(T:Transaction)
precondition:
    T ∈ self.commit_requested;
postcondition:
    self.committed= self.committed ∪{T};
    self.readlock_holders = self.readlock_holders - T;
    self.writelock_holders = self.writelock_holders - T.

Scheduler::Abort(T:Transaction)
precondition:
postcondition:
    self.aborted= self.aborted ∪{T};
    self.readlock_holders = self.readlock_holders - T;
    self.writelock_holders = self.writelock_holders - T.
```

Figure 23. Specification of the interface of Scheduler based on Locking

There are five operations, *HandleOperation* for read, *HandleOperation* for write, *CommitRequest*, *Commit* and *Abort*. The operation *HandleOperation* checks whether the operation can be abstracted to a read or write operation. This means that the original operation does not need to be a read or write at all. Subsequently, a check is done on whether the corresponding operation conflicts with previously submitted operations of other transactions. The conflict operation is hereby encapsulated and may depend on different conflict rules for different operations. Before a *Commit* operation can occur first a *CommitRequest* operation must be invoked. A *CommitRequest* operation may also conflict with other operations and therefore this is also explicitly checked. The *Commit* and *Abort* operations result in the release of the locks that have been hold in the sets *readlock_holders* and *writelock_holders* of the corresponding transaction are released.

For the same architectural component *Scheduler* we may derive other semantics from the solution domain on concurrency control. Figure 24 defines, for instance, the specification of the operations of a timestamp ordering scheduler [Bernstein et al. 87].

```
Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "read"
precondition:
    there do not exist (T',n) such that if (T',n) conflicts with (T,m)
    and timestamp(T') > timestamp(T)
postcondition:
    self.max_readtimestamp = timestamp(T);

Scheduler::HandleOperation(T:Transaction, m: Operation)
for kind(m) = "write"
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m)
    and timestamp(T') > timestamp(T)
postcondition:
    self.max_writetimestamp = timestamp(T)

Scheduler::CommitRequest(T:Transaction)
precondition:
    there do not exist (T',n) such that (T,m) conflicts with (T,m);
postcondition:
    self.commit_requested = self.commit_requested ∪ T;

Scheduler::Commit(T:Transaction)
precondition:
    T ∈ self.commit_requested;
postcondition:
    self.committed= self.committed ∪{T};

Scheduler::Abort(T:Transaction)
precondition:
postcondition:
    self.aborted= self.aborted ∪{T};
```

Figure 24. Specification of the interface of Scheduler based on timestamp ordering

The timestamp ordering scheduler orders conflicting operations according to their timestamps that have been assigned by *TransactionManager*. If two operation p and q are conflicting then the timestamp ordering scheduler processes p before q if *timestamp*(p) < *timestamp*(q).

RecoveryManager

Figure 25 represents a specification of the interface of RecoveryManager that has been adapted from the solution domain on recovery [Bhargava 87]. In this example, the RecoveryManager has 5 operations. The operation *HandleOperation* will either log the operation or the state of the data object that is being accessed. The operation *Commit* makes the effect of the transaction persistent by storing this in stable storage. The operation *Abort* rollbacks the effects of the transaction by using the logged information. The operation *Restart* will be invoked in case of system failures. This operation uses the logged information to undo the effects of the aborted or active transactions and redo the effects of the committed transactions that have not been made persistent yet. Finally, the operation *Checkpoint* is regularly invoked to make a snapshot of the system so that the *Restart* operation is optimized. In Figure 25 only a generic interface for recovery is presented. However, the semantics for each of the different variations on these recovery protocols can be easily derived from the solution domains.

RecoveryManager::HandleOperation(T:Transaction, m: Operation)
postcondition:
 operation (T,m) or the accessed object state is logged;

RecoveryManager::Commit(T:Transaction)
postcondition:
 make effects of transaction T persistent;

RecoveryManager::Abort(T:Transaction)
postcondition:
 undo effects of transaction T;

RecoveryManager::Restart()
postcondition:
 undo effects of active aborted transactions
 redo effects of committed transactions;

RecoveryManager::Checkpoint()
postcondition:
 store the state of the system in stable storage.

Figure 25. Specification of the operations of RecoveryManager

PolicyManager

An example specification for (a part of) the interface of the PolicyManager component is given in Figure 26.

PolicyManager::AddParameter(P: PerformanceParameter)
postcondition:
 self.performanceParameters = self.performanceParameters $\cup$ P;

PolicyManager::RemoveParameter(P: PerformanceParameter)
postcondition:
 self.performanceParameters = self.performanceParameters - P;

PolicyManager::ReadParameterValues()
postcondition:
 self.performanceParameters determined;

PolicyManager::ChooseTransactionProtocols(T:Transaction)
postcondition:
 T.transactionProtocols determined;

PolicyManager::DeterminePolicy()
postcondition:
 self.policy = *priority*(user, system, data).

Figure 26. Specification of the interface of PolicyManager

PolicyManager is responsible for dynamic adaptation of the transaction protocols based on selected performance parameters such as transaction throughput, transaction response time, transaction blocking ratio and transaction restart ratio [Agrawal 87]. The operations *AddParameter* and *RemoveParameter* respectively add and remove a performance parameter from the set *performanceParameters*. The operation *ReadSystemParameters* senses the system and derives the values for the parameters in the set *performanceParameters*. These values are used by the operation *ChooseTransactionProtocols* to determine appropriate transaction protocols such as scheduling and recovery algorithms. Finally, the operation *DeterminePolicy* defines the policy for the dynamic adaptation mechanism. The choice of transaction protocols may not always be defined by the system characteristics but the transaction or data characteristics may also impose some constraints on the selection of the transaction protocols. For example, for long transaction a locking scheduler may be preferred over an optimistic scheduler [Agrawal 87]. Large binary data objects may prefer to adopt operation logging techniques instead of image logging to optimize the memory space [Elmagarmid 92]. *PolicyManager* must therefore balance between these different wishes of the transaction programmer, the system performance parameters and the data object characteristics.

Correctness of Transaction Semantics

We have shown that the semantics for the components of the atomic software architecture can be derived from the solution domains and gave some examples of the semantics of the architectural components of the atomic transaction architecture. Besides of the rich semantics that we could derive from the solution domain an important issue is whether the provided semantics is correct. In this thesis we will not provide the correctness proofs but refer to the related literature on atomic transactions. For example, in [Lynch et al. 94] the I/O automaton model is described, which is a formal model for modeling concurrent, and distributed systems. Hereby, each system component, concept or technique is analyzed and expressed as an automaton, a mathematical object with states and named transitions between them[8]. The actions of the automaton can be classified as *input*, *output* or *internal*. The input actions represent events from the environment, the output actions represent events that components performs itself and finally the internal actions represent the events internal in a component that are not externally observable (such as changing a local variable). For each automaton the

[8] This is almost similar to a non-deterministic finite-state automaton. One difference is that in the I/O automaton model an automaton need not be finite-state, but can have an infinite state set.

actions are described with an *action signature*. An automaton can put restrictions on when it will perform an output or internal action, but is unable to restrict input actions.

The correctness criteria for a concurrent system that is modeled as automata are expressed as restrictions on the sequences of actions that are part of the interface of the data items and its users. The basic assumption is that a sequence of actions is correct if it can be generated by a serial system.

An automaton is defined as a tuple consisting of four components [Lynch et al. 94]:

- an action signature *sig(A)*
- a set *states(A)* of **states**
- a nonempty set *start(A)* $\subseteq$ *states(A)* of **start states**, and
- a **transition relation** *steps(A)* $\subseteq$ *states(A)* x *acts(sig(A))* x *states(A)*, with the property that for every state s' and input action π there is a transition (s', π, s) in *steps(A)*.

In [Lynch et al. 94] states are generally determined by giving values to a collection of variables. Further, the transition relations of an automaton are not described by listing all its elements as triples but rather a simple specification language is used where an *effect* is described for each action and a *precondition* for each local action.

Using this model the authors formalize and analyze transaction processing theories, serializability, logging, locking, nesting, timestamping etc. For a more detailed description of this automaton model we refer to [Lynch et al. 94]. We use this model to proof the semantics of the architectural concepts that we derived. It follows that since we derive the abstract semantics from the solution domain, the link to the formal models is easily identified and we can utilize these to validate the software architecture. For this, we map each architectural concept to an automaton and define the operations in the specification of the concept as *internal* and *input* actions of the newly identified automaton. To define the *output* actions we look for the architectural concepts that the corresponding concept communicates with and define the operations of other concepts that are invoked as the output operations of the automaton. These operations together will provide the complete action signature of the automaton. Consider for example the specification of the concept *TransactionManager* as presented in Figure 21. We could define an automaton called *TransactionManager* that includes the operations as defined in Figure 21. Since all of these operations are invoked by other components we map these to *input* operations of the automaton. There are no internal operations. The output operations can be identified in the specification of the concept *Transaction* in Figure 20 and this completes the action signature of the automaton *TransactionManager*. Subsequently the start states of the automaton and the transition relations can be relatively easily defined. Once the automaton is described we use it to continue with the correctness proofs of the adopted transaction semantics. Since many publications exist on correctness proofs of the semantics of transaction protocols [Papadimitriou 86], [Cellary et al. 89] and [Bernstein et al. 87] and it is not our goal to extend the transaction theory we will not elaborate on this issue in this thesis.

Define Dynamic Behavior of the Architecture

The specifications of the architectural components are used to model the dynamic behavior of the architecture. For this purpose we use the so-called collaboration diagrams which are interaction diagrams to illustrate the dynamic view of a system [Booch et al. 99]. Collaboration diagrams show the structural organization of the components and the interaction among these components. We derive the collaboration diagrams from the pre-defined specifications of the architectural concepts.

Example

Figure 27 represents an example of a collaboration diagram for the atomic transaction architecture. The components in the collaboration diagram represent instances of the architectural components, which is represented by a double colon preceding the name of the architectural component. The flow of control is represented by means of directed arrows that are labeled with messages. To indicate the temporal sequencing the messages are numbered. The collaboration diagram shows the interactions for starting, handling operations, committing and aborting transactions.

The messages with the sequence number 1 are part of the scenario for starting a transaction. A transaction is started by the object t:TransactionApplication that sends a start operation to the object tm:TransactionManager. The tm object informs the starting of the new transaction to the object pm:PolicyManager that reads the values of the performance parameters and chooses the appropriate transaction protocols for the transaction.

The messages with sequence numbers 2 define a scenario for handling transaction operations. After the transaction has started, the operations that are send by the object t:aTransactionApplication will be captured and handled by the object tm:TransactionManager. The operation will be forwarded to the policy manager and the data manager object. The object dm:DataManager will request the scheduler and the recovery manager object to provide a decision on the acceptance or reject of the operation. If the operation is allowed to execute then it will be dispatched to the atomic object.

Finally, the messages with the sequence numbers 3 and 4 define respectively the scenarios for committing and aborting transactions. The control flow for these scenarios can be easily derived from Figure 27.

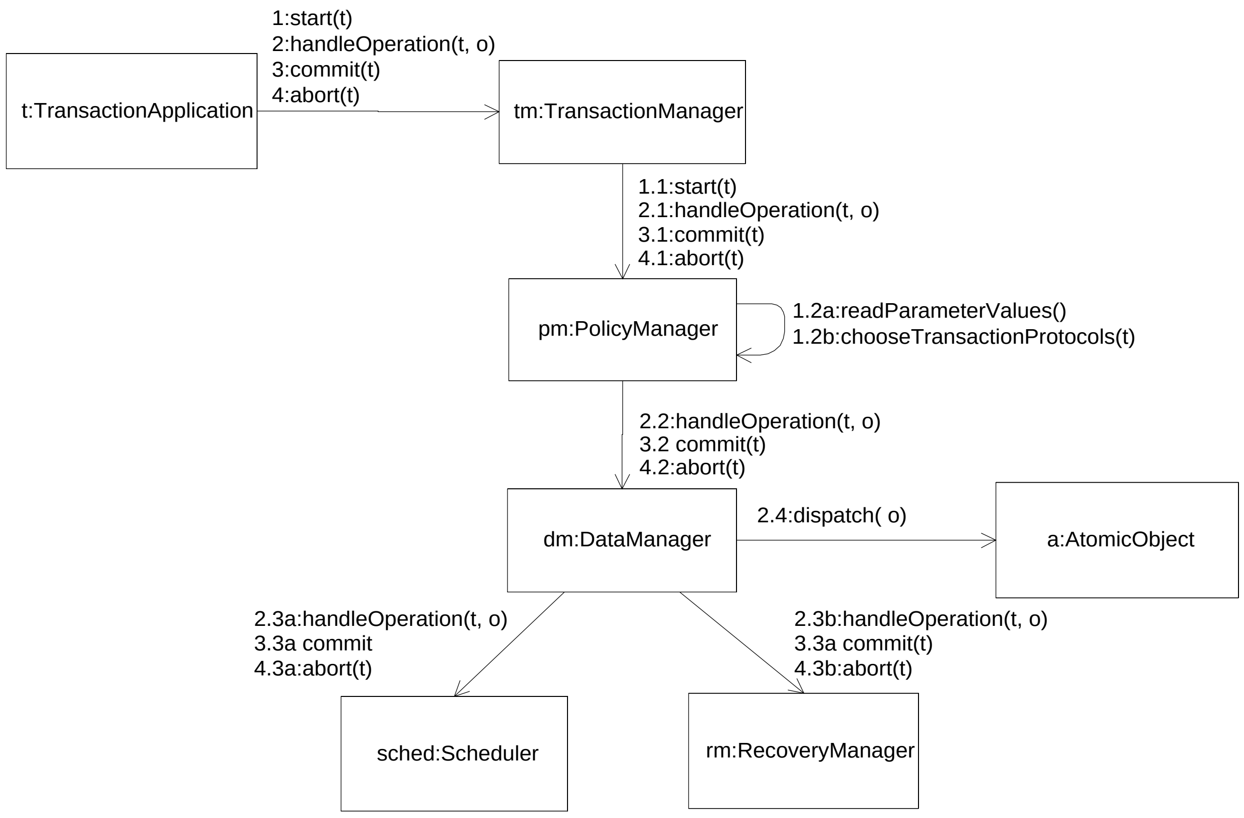

Figure 27. Collaboration diagram for the atomic transaction software architecture

4.5 Discussion and Conclusions

In this chapter we presented the *synthesis-based software architecture design approach.* This approach is derived from the concept *synthesis* of mature engineering disciplines whereby the initial problem is decomposed into sub-problems that are solved separately and later integrated in the overall solution. During the synthesis process design alternatives are searched and selected based on the existing solution domain knowledge.

An important issue in software architecture design is to find the right abstractions and the adequate leveraging of the architecture. The novelty of the *synthesis-based software architecture design approach* with respect to the existing architecture design approaches is that it makes the processes of *problem analysis*, *solution domain analysis* and *alternative space analysis* explicit. During the *problem analysis*, the client requirements are mapped onto the technical problems providing a more objective and reliable description of the problem. During the *solution domain analysis*, stable architectural components with rich semantics are derived from the solution domain concepts that are well-defined and stable themselves. The solution domain analysis itself is leveraged by the pre-identified technical problems so that the right detail of the solution domain model is ensured. The *alternative space analysis* explicitly depicts the possible set of design alternatives that can be derived from the architectural components.

We have illustrated the approach by applying it to the design of an atomic transaction architecture for a distributed car dealer system in a project of Siemens-Nixdorf. Apart from this, experimental studies have been carried out with earlier versions of this approach in pilot studies that were carried out by MSc students. For example, in [Vuijst 94], a software architecture for image algebra was derived for the laboratory for clinical and experimental image processing. The basic solution domain for this architecture was image algebra and several related publications could be identified from which sufficient stable abstractions were derived for the design of the software architecture. The atomic transaction and the image algebra domain appeared to be examples of well-defined and sufficiently formalized domains. The experimental studies have been, however, also applied on domains that are less formalized. In [Arend 99], for example, a software architecture has been derived for a Quality Management Systems for efficient information retrieval and in [Willems 98], a software architecture has been derived for insurance systems. In both cases, several publications could be identified on the corresponding domains, but in addition it was also necessary to refer to the factual knowledge and experiences for the design of the software architecture. The solution domain may thus consist of a combination of various forms of solution techniques such as theories, solution domain experts, and experiences in the corresponding domain.

In the following we will list the conclusions that we could obtain from our experience in applying the synthesis approach to the project on atomic transactions.

1. *Explicit mapping of requirements to technical problems facilitates the identification and leveraging of the necessary solution domains.*

After our requirements analysis and technical problem analysis processes as defined in sections 4.4.1 and 4.4.2 respectively, it appeared that the given client requirements did not fully describe the right detail of the desired problem. The basic requirement was to provide adaptable transactions protocols that were derived from the various expected needs of different dealers in different countries. From the initial requirement specification, however, it followed that with adaptability of transaction protocols it was only referred to a restricted number of concurrency control protocols. During the problem analysis phase we generalized this requirement to the adaptation of various transaction protocols including transaction management, concurrency control, recovery and data management techniques. After interactions with the client and a study of the car dealer distribution system it appeared that many transaction protocols were relevant although they had not been explicitly mentioned in the requirement specification. We observed that the technical problem identification is an iterative process between the technical problem analysis and solution domain analysis processes.

On the one hand, we directed and leveraged our solution domain analysis using the identified technical problems. Since every (sub-)problem corresponds only to a restricted set of solution domain we did not need to consider the whole solution domain space at once. For example, for the concept *DataManager* we did not need to consider version management and replication management because this was deliberately put out of the scope of the project. For the concept *Scheduler* we ruled out the solution domain that dealt with semantic concurrency control techniques. The identified technical problems provided us the means where to search or not to search for the solution domain.

On the other hand, the technical problems could be better defined after the solution domains were better understood. For example, only after a solution domain analysis on concurrency control, as described in section 4.4.3, we were better able to accurately define the sub-problems related with the concept *Scheduler*. This observation may imply that for the problem analysis phase one may require a domain engineer who is an expert on the corresponding domain and knows the different technical problems that are related to the domain. In our example project typically a transaction domain expert at the early phase of problem analysis would be of much help.

2. Solution domain provides stable architectural abstractions

The synthesis-based approach provides an explicit solution domain analysis process for identifying the right abstractions. After analyzing and comparing the solution domain on transaction theory it appears that it is rather stable and does not change abruptly but only shows a gradual specialization of the transaction concepts. Because the solution domain is stable it provides a reliable source for providing stable architectural abstractions. In the solution domain analysis process as described in section 4.4.3 we illustrated how we could derive stable concepts for the design of the atomic transaction architecture. We were able to derive both the overall architecture and refine the architectural concepts to the required detail level.

The requirement of stable solution domains in the synthesis-based approach implies that a given problem can only be solved to the extent that it has been explored in the solution domain. If it appears that the solution domain is not well-established the software engineer may decide to terminate the synthesis process, reformulate the technical problem or initiate a research on the solution domain. The latter decision shows that the synthesis process may provide important input for the scientific research because it may indicate the issues that need to be resolved in the corresponding solution domains[9].

3. *Solution domains provide rich semantics for realization and verification of the architecture.*

Solution domains not only provide stable abstractions but in addition these abstractions have rich semantics which is important for the realization and verification of the software architecture. As described in section 4.4.5 on architecture specification, we could derive rich semantics for the architectural abstractions directly from the solution domain knowledge of atomic transactions. We have illustrated this process for various components in the atomic transaction architecture.

The solution domain is not only useful for deriving architectural abstractions, but in addition it is also a reliable source for validating the correctness of the developed architecture. We were able to identify many publications that explicitly deal with correctness proofs of various transaction protocols. We validated the architectural components and their semantics by utilizing these knowledge sources.

4. *Adaptability of an architecture can be determined by an explicit alternative space analysis of the solution domain.*

In the synthesis-based software architecture design approach, alternative space analysis is an explicit process. Thereby, for each concept the set of alternatives are described and constraints are defined among these alternatives. This together results in a depiction of the set of possible alternative designs or alternative space, that may be derived from the given software architecture. As described in section 4.4.4 we have, for instance, defined the alternatives for the concepts *Scheduler* and *RecoveryManager*. From the solution domain analysis we extracted the constraints within each of these concepts and constraints that apply among alternatives of these concepts. We had two problems in the alternative space analysis process for the example project. First, although we have derived the conceptual architectures from the solution domain itself, during the alternative definition process it followed that not all the alternatives were explicitly described in the literature. For example, for the concept *Scheduler* we could identify only around 10-15 scheduler types that were described in the literature. The other alternatives are primarily seen as variations of these basic scheduler types. In our approach we could depict every single alternative explicitly. The second problem that we encountered was that the constraints within and among the alternatives of the concepts are generally not explicitly stated in the literature and finding these is very time-consuming. Defining constraints of solution domain concepts requires the full understanding of these concepts. The existence of an explicit description of these constraints may indicate the maturity level of the corresponding solution domain. It appears

[9] Note that this represents an example of the interaction between engineering and scientific research

that the transaction literature has many well-established concepts and we could also identify some publications that explicitly dealt with the constraints among the concepts, however, this is not the case for all the concepts.

4.6 References

[Agrawal 87] Agrawal, R., Carey, M., & Livney, M. *Concurrency control performance modelling: Alternatives and implications.* ACM Transactions on Database Systems, Vol. 12, No. 4, pp. 609-654, December 1987.

[Ahsmann & Bergmans 95] Ahsmann F & Bergmans L. *I-NEDIS: New European Dealer System*, Project plan I-NEDIS, 1995.

[Aksit 00] Aksit, M. *Course Notes: Design Software Architectures.* Post-Academic Organization, 2000.

[Aksit et al. 98] Aksit, M., Marcelloni, F., & Tekinerdogan B. *Developing Object-Oriented Frameworks using domain models*, ACM computing surveys, 1998.

[Aksit et al. 99] Aksit, M., Tekinerdogan, B., Marcelloni, F., & Bergmans, L. *Deriving Object-Oriented Frameworks from Domain Knowledge.* in M. Fayad, D. Schmidt, R. Johnson (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley & Sons, 1999.

[Aksit et al. 96] Aksit, M., Tekinerdogan, B, & Bergmans, L. *Achieving adaptability through separation and composition of concerns*, in Max Muhlhauser (ed), Special issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP '96, Linz, Austria, July, 1996.

[Arend 99] Arend, E. van der. *Design of an Architecture for a Quality Management Push Framework.* MSc thesis, Dept. of Computer Science, University of Twente, 1999.

[Arrango 94] Arrango, G. *Domain Analysis Methods.* In *Software Reusability,* Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.

[Atkins & Coady 92] Atkins, M.S. & Coady, M.Y. *Adaptable Concurrency Control for Atomic Data Types.* ACM Transactions on Computer Systems, Vol. 10, No. 3, pp. 190-225, August 1992.

[Barghouti & Kaiser 91] Barghouti, N.S., & Kaiser, G.E. *Concurrency Control in Advanced Database Applications,* ACM Computing Surveys, Vol. 23, No. 3, September, 1991.

[Bass et al. 98] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice,* Addison-Wesley 1998.

[Bernstein & Goodman 83] Bernstein, A., & Goodman, N. *Concurrency Control in Distributed Database Systems,* ACM Transactions on Database Systems, 8(4): 484-502, 1983.

[Bernstein & Newcomer 97] Bernstein, P.A., & Newcomer, E. *Principles of Transaction Processing,* Morgan Kaufman Publishers, 1997.

[Bernstein et al. 87] Bernstein, P.A., Hadzilacos, V., & Goodman, N. *Concurrency Control & Recovery in Database Systems,* Addison Wesley, 1987.

[Bhargava 87] Bhargava, B.K. editor. *Concurrency Control and Reliability in distributed Systems,*Van Nostrand Reinhold, 1987.

[Booch et al. 99] Booch, G., Jacobson, I., & Rumbaugh, J. *The Unified Modeling Language User Guide,* Addison-Wesley, 1999.

[Bourque et al. 99] Bourque, P., Dupuis, R., Abran, A., Moore, J.W., & Tripp, L. *The Guide to the Software Engineering Body of Knowledge,* Vol. 16, No. 6, pp. 35-45, November / December, 1999.

[Buschmann et al. 99] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1999.

[Carey 84] Carey, M., & Stonebraker, M. *The performance of concurrency control algorithms for database management systems*. In Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, pp. 107-118, 1984.

[Cellary et al. 89] Cellary, W., Gelenbe, E., & Morzy, T. *Concurrency Control in Distributed Database Systems*, North-Holland Press, 1989.

[Coyne et al. 90] Coyne, R.D., Rosenman, M.A., Radford, A.D., Balachandran, M., & Gero, J.S. *Knowledge-Based Design Systems*, Addison-Wesley, 1990.

[Cross 89] Cross, N. *Engineering Design Methods*, Wiley and Sons, 1989.

[Date 90] Date, C.J. *An Introduction to Database Systems*, Vol. 3, Addison Wesley, 1990.

[Diaper 89a] Diaper, D. (ed.). *Knowledge Elicitation*, Ellis Horwood, Chichester, 1989.

[Diaper 89b] Diaper, D. *Task Analysis for Human Computer Interaction*, Wiley & Sons, 1989.

[Dorf & Bishop 98] Dorf, R.C., & Bishop, R.H. *Modern Control Systems*. Addison-Wesley, 1998.

[Elmagarmid 92] Elmagarmid, A.K. editor. *Database Transaction Models for AdvancedApplications Transaction Management in Database Systems*, Morgan Kaufmann Publishers, 1992.

[Firlej & Hellens 91] Firlej, M., & Hellens, D. *Knowledge elicitation: a practical handbook*, New York, Prentice Hall, 1991.

[Foerster 79] Foerster, H. Von., *Cybernetics of Cybernetics*, in: Klaus Krippendorff (ed.), Communication and Control in Society, New York: Gordon and Breach, 1979.

[Gajski et al. 92] Gajski, D.D., Dutt, N.D., Wu, A., & Lin, S. *High-level synthesis : introduction to chip and system design*, Boston : Kluwer Academic Publishers, 1992.

[Glass & Vessey 95] Glass, R.L., & Vessey, I. *Contemporary Application-Domain Taxonomies*, IEEE Software, Vol. 12, No. 4, July 1995.

[Gonzales & Dankel 93] Gonzalez, A.J., & Dankel, D.D. *The Engineering of Knowledge-Based Systems*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[Gray & Reuter 93] Gray, J., & Reuter, A. *Transaction processing: concepts and techniques*, San Mateo, Morgan Kaufmann Publishers 1993.

[Guerraoui 94] Guerraoui, R. *Atomic Object Composition*. In Proceedings of the European Conference on Object-Oriented Programming, LNCS 821, Springer-Verlag, pp. 118-138, 1994.

[Hadzilacos 88] Hadzilacos, V. *A theory of reliability in Database Systems*, Journal of the ACM, 35(1): 121-145, January 1988.

[Haerder & Reuter 83] Haerder, T., & Reuter, A. *Principles of Transaction-Oriented Database Recovery*. ACM Computing Surveys, Vol. 15. No. 4. pp. 287-317, 1983.

[Highleyman 89] Highleyman, W.H. *Performance analysis of transaction processing systems*, Englewood Cliffs, NJ : Prentice Hall, 1989.

[Howard 87] Howard, R.W. *Concepts and Schemata: An Introduction*, Cassel Education, 1987.

[Jacobson et al. 99] Jacobson, I., Booch, G., & Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jajodia & Kerschberg 97] Jajodia, S., & Kerschberg, L. *Advanced Transaction Models and Architectures*, Boston: Kluwer Academic Publishers, 1997.

[Kruchten 95] Kruchten, Philippe B. *The 4+1 View Model of Architecture*. IEEE Software, Vol 12, No 6, pp. 42-50, November 1995.

[Kumar 96] Kumar, V. *Performance of Concurrency Control Mechanisms in Centralized Database Systems.* Prentice-Hall, 1996.

[Lakoff 87] Lakoff, G. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind,* The University of Chicago Press, 1987.

[Lieberherr 96] Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns,* PWS Publishing Company, Boston, 1996.

[Loucopoulos & Karakostas 95] Loucopoulos, P., & Karakostas, V. *System requirements engineering,* London [etc.] : McGraw-Hill, 1995.

[Lynch et al. 94] Lynch, N., Merrit, M., Weihl, W., & Fekete, A. *Atomic Transactions.* Morgan Kaufmann Publishers, 1994.

[Maher 90] Maher, M.L., *Process Models for Design Synthesis,* AI-Magazine, pp. 49-58, Winter 1990.

[Maimon & Braha 96] Maimon, O., & Braha, D. *On the Complexity of the Design Synthesis Problem*, IEEE Transactions on Systems, Man, And Cybernetics-Part A: Systems and Humans, Vol. 26, No. 1, January 1996.

[Meyer & Booker 91] Meyer, M., & Booker, J. *Eliciting and Analyzing Expert Judgment: A practical Guide, Volume 5 of Knowledge-Based Systems,* London: Academic Press, 1991.

[Moss 85] Moss, J.E.B. *Nested Transactions : an approach to reliable distributed computing,* Cambridge, MA: MIT Press, 1985.

[Newell & Simon 76] Newell, A., & Simon, H.A., *Human Problem Solving,* Prentice-Hall, Englewood Clifss, NJ, 1976.

[Papadimitriou 86] C.H. Papadimitriou. *The theory of Database Concurrency Control.* Computer Science Press, 1986.

[Parsons & Wand 97] Parsons, J., & Wand, Y. *Choosing Classes in Conceptual Modeling,* Communications of the ACM, Vol 40. No. 6., pp. 63-69, 1997

[Partridge & Hussain 95] Partridge, D., & Hussain, K.M. *Knowledge-Based Information Systems,* McGraw-Hill, 1995.

[Prieto-Diaz & Arrango 91] Prieto-Diaz, R., & Arrango, G. (Eds.). *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, Los Alamitos, California, 1991.

[Polya 57] Polya, G. *How to solve it : a new aspect of mathematical method,* New York, Doubleday, 1957.

[Pu et al. 88] Pu, C., Kaiser, G., & Hutchinson, N. *Split-transactions for open-ended activities.* In Proceedings of the 14th VLDB Conference, 1988.

[Reich & Fenves 91] Reich, Y., & Fenves, S.J. *The formation and use of abstract concepts in design,* in: Concept Formation: Knowledge and Experience in Unsupervised Learning, D.H.J. Fisher, M.J. Pazzani, & P. Langley (eds.), Los Altos, CA, pp. 323--353, Morgan Kaufmann, 1991.

[Roxin 97] Roxin, E.O. *Control theory and its applications.* Amsterdam, Gordon and Breach Science Publishers, 1997.

[Rubin 98] Rubin, R. *Foundations of library and information science.* New York, Neal-Schuman, 1998.

[Shaw & Garlan 96] Shaw, M. & Garlan, D. *Software Architectures: Perspectives on an Emerging Discipline,.* Englewood Cliffs, NJ: Prentice-Hall, 1996.

[Shaw 98] Shaw, M. *Moving from Qualities to Architectures: Architectural Styles,* in: L. Bass, P. Clements, & R. Kazman (eds.), Software Architecture in Practice, Addison-Wesley, 1998.

[Shinners 98] Shinners, S.M. *Modern Control System Theory and Design.* Wiley, 1998.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction.* Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Sommerville & Sawyer 97] Sommerville, I., & Sawyer, P. *Requirements engineering: a good practice guide*, Chichester, Wiley, 1997.

[Stillings et al. 95] Stillings, N.A., Weisler, S.E., Chase, C.H., Feinstein, M.H., Garfield, J.L., & Rissland, E.L., *Cognitive Science: An Introduction.* Second Edition, The MIT Press, Cambridge, Massachusetts, 1995.

[Tekinerdogan 94] Tekinerdogan, B. *Design of an object-oriented framework for atomic transactions*, MSc. thesis, University of Twente, Dept of Computer Science, 1994.

[Tekinerdogan 95a] Tekinerdogan, B., *Overall Requirements Analysis for INEDIS*, Siemens-Nixdorf/University of Twente, INEDIS project, 1995.

[Tekinerdogan 95b] Tekinerdogan, B. *Requirements for Transaction Processing in INEDIS*, Siemens-Nixdorf/University of Twente, INEDIS project, 1995.

[Tekinerdogan 96] Tekinerdogan, B. *Reliability problems and issues in a distributed car dealer information system*, INEDIS project, 1996.

[Tekinerdogan & Aksit 97] Tekinerdogan, B., & Aksit, M. *Adaptability in object-oriented software development*, Workshop report, in M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.

[Tekinerdogan & Aksit 99] Tekinerdogan, B., & Aksit, M. *Deriving design aspects from conceptual models.* In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 410-414, 1999.

[Tracz & Coglianese 92] W. Tracz and L. Coglianese. *DSSA Engineering Process Guidelines. Technical Report*, ADAGE-IBM-9202, IBM Federal Systems Company, December, 1992.

[Thayer et al. 97] Thayer, R.H., Dorfman, M., & Bailin, S.C. *Software requirements engineering*, Los Alamitos, IEEE Computer Society Press, 1997.

[Traiger et al. 82] Traiger, I.L., Gray, J., Caltiere, C.A., & Lindsay, B.G. *Transactions and Consistency in Distributed Database Systems*, ACM Transactions on Database Systems, Vol. 7, No. 3, September 1982, pp 323-342.

[Umplebey 90] Umplebey, S.A., *The Science of Cybernetics and the Cybernetics of Science*, Cybernetics and Systems, Vol. 21, No. 1, 1990, pp. 109-121, 1990.

[Vuijst 94] Vuijst, C. *Design of an Object-Oriented Framework for Image Algebra*. MSc thesis, Dept. of Computer Science, University of Twente, 1994.

[Warmer & Kleppe 99] Warmer, J.B., & Kleppe, A.G. *The Object Constraint Language : Precise Modeling With Uml*, Addison-Wesley, 1999.

[Wartik & Prieto-Diaz 92] Wartik, S., & Prieto-Díaz, R. *Criteria for Comparing Domain Analysis Approaches*. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, pp. 403-431, September 1992.

[Weihl 89] Weihl, W. *The impact of recovery on concurrency control*. Proceedings of the eigth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems March 29 - 31, Philadelphia, PA USA, 1989.

[Weihl 90] Weihl, W.E. *Linguistic support for atomic data types*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 2, 1990.

[Wielinga et al. 92] Wielinga, B.J., Schreiber, T., & Breuker, J.A., *KADS: a modeling approach to knowledge engineering*, Academic Press, 1992.

[Willems 98] Willems, R. *Ontwikkelen van verzekeringsproducten*, dutch, translation: Development of Insurance Products, MSc thesis, Dept. of Computer Science, University of Twente, 1999.

[Wu et al. 95] Wu, Z., Stroud, R.J., Moody, K., & Bacon, J. *The design and implementation of a distributed transaction system based on atomic data types*, Distributed Syst, Engineering, 2, pp. 50-64, 1995.

Table of Contents