Chapter 5

SYNTHESIS-BASED SOFTWARE ARCHITECTURE DESIGN

Bedir Tekinerdoğan and Mehmet Aksit
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: [bedir, aksit]@cs.utwente.nl,
www: http://trese.cs.utwente.nl

Keywords: Software architecture design, Synthesis, Domain analysis, Problem-Solving, Synbad

Abstract: Software architectures provide the gross-level design and as such impact the quality of the entire system. To support the quality factors such as robustness, adaptability and maintainability, a proper scoping of the architecture boundaries and likewise the identification of the relevant architectural abstractions is necessary. Several architecture design approaches have been introduced whereby the scoping of the architecture is merely based on the stakeholder's perspective. This chapter introduces a novel software architecture design approach that aims to scope the architecture boundaries from a systematic problem-solving perspective instead. In this so-called synthesis-based architecture design approach (Synbad), the client's perspective is abstracted to derive the technical problems. The technical problems define the scope of the solution domains from which the architectural abstractions are derived. The approach is illustrated for the design of an atomic transaction architecture for a real industrial project.

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 [6][15][37][50]. However, a consensus on the appropriate software architecture design process is not established yet and...
current software architecture design approaches 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. In contrast, 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 merely from the client's-perspective rather than on the architectural solution perspective of the system. The gap between the client perspective and the architectural design perspective, however, is generally too large and the client may lack to specify the right detail of the problem. Due to the inappropriate scoping of the problem the fundamental transparent abstractions may be missed and/or redundant abstractions may be elicited.

Third, the adopted sources from the client's perspective are not very useful in providing sufficiently rich semantics of the architectural components and in providing guidelines for composing the architectural abstractions. In this case, architectural components are often equivalent to semantically poor groupings.

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 termed synthesis-based software architecture design, Synbad for short, is proposed, which aims providing effective solutions to these problems. In this 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. Finally, the individual sub-solutions are synthesized in the overall software architecture. The novelty of this approach

---

1 Chapter 1 of this book provides an in-depth analysis of the current architectural design methods.

2 We use the term client to denote any stakeholder who has interest in the application of a software architecture.
is that it explicitly integrates the processes of technical problem analysis, solution domain analysis and alternative design space analysis.

The approach will be demonstrated using a project on the design of an atomic transaction system architecture for a distributed car dealer information system.

The remainder of the chapter is organized as follows. In section 2, the synthesis concept is described and a model for software architecture synthesis is derived. In section 3, an example project on the design of a software architecture for atomic transactions for a distributed car dealer information system will be described, which will be used throughout the whole chapter. In section 4, Synbad will be presented that will be illustrated for the example project. Finally, in section 5, we will present our discussion and conclusions.

2. SYNTHESIS

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 [35]. A well-known and widely applied problem solving technique in traditional engineering disciplines such as electrical engineering, chemical engineering and mechanical engineering is the concept of synthesis [37]. In section 2.1 we will explain this concept of synthesis as it is described in traditional engineering disciplines. In section 2.2 we will provide a software architecture synthesis model that represents the integration of the synthesis concept in software architecture design and as such forms a basis for Synbad, the synthesis-based software architecture design approach.

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. In particular, the synthesis process includes an explicit phase for searching solution domains, searching design alternatives in the corresponding solution domain and selecting these alternatives based on explicit quality criteria.

---

1 This work has been carried out as part of the INEDIS project that was a collaborative project between Siemens-Nixdorf and the TSE group, Software Engineering, Dept. of Computer Science, University of Twente.
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 [38]. 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 the design solutions so far (design state) are evaluated with respect to its consistency 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 a complete analysis is impossible in any but the simplest problems [17]. Therefore, in practice, 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 [17].

In [38] 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 [23]. 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 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 consist of the fixed configuration of lower level abstractions thereby implicitly reducing the alternative configuration possibilities. This is acceptable, though, since usually the total space of a synthesis from higher level abstractions is large enough to be of practical use.

2.2 Software Architecture Synthesis Model

Figure 1 represents a conceptual model for software architecture synthesis [35]. This model has been derived from our experiences in designing software architectures for various applications [3][5][63][67]. In addition it conforms to the synthesis-based design as it is widely accepted in mature engineering disciplines. Basically, it aims to explicitly integrate the processes of technical problem analysis, solution domain analysis, and alternative space analysis.

![Figure 1: Architecture synthesis model](image-url)
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. Note that this model represents a conceptual view of the software architecture synthesis process and does not enforce specific control flows between the various processes. In the following we will explain the concepts and functions of both parts of the model.

2.2.1 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 explicitly separates the concepts Requirement Specification and Technical Problem.

The function Formulate utilizes the Requirement Specification to define 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 Discover represents the process of discovering new sub-problems when new solution abstractions are extracted from the solution domain knowledge.
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 Impact represents the process of refining the requirement specification from the results of the architecture specification.

2.2.2 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. These are derived from the technical problems and/or the solution domain knowledge.

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 using the quality criteria and the constraints. 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.

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. A basic requirement was the automated support for processes such
as workshop processing, order processing, stock management, new and used
car management, and financial accounting. Further, the car dealer system
required the execution of the tasks consistently and effectively. The meaning
of consistency, in general, 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.

In a distributed system as the car dealer system, there are two main
factors that threaten the consistency of data: 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. In addition it is generally
demanded that the implementation of these concurrency and recovery
mechanisms is transparent to the application program developers, since they
will need only the primitives and do not 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, and was to 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.

In addition to the need for adaptability at initialization time, the system
required also adaptation at run-time. The car dealer system is 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. **SYNBAD: SYNTHESIS-BASED SOFTWARE ARCHITECTURE DESIGN PROCESS**

In this section, the synthesis-based software architecture design process that implements the process of the Architecture Synthesis Model of Figure 1 will be described.

The following sections are organized around the basic processes of the approach. Section 4.1 describes the *Requirements Analysis* process, section 4.2 the *Problem Analysis* process, section 4.3 the *Solution Domain Analysis* process, section 4.4 *Alternative Space Analysis* process and finally section 4.5 the *Architecture Specification* process.

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. [28]. The requirements analysis process concerns the concept *Requirement Specification* of Figure 1.

In Synbad the well-known requirement analysis techniques such as informal requirement specifications, use-cases [21] and scenarios [32], constructing prototypes and defining finite state machine modeling are applied. Informal requirement specification serves 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 these in this chapter and refer for detailed information to the corresponding publications [59][51][35].
4.2 Technical Problem Analysis

The requirements analysis process provides an understanding of the client perspective of the software system. In the technical problem analysis process the identified client requirements are mapped to technical problems. The underlying motivation for this technical problem analysis process is the idea that software architecture is in essence a problem solving process in which the solution represents an architecture design. In this sense, the technical problem analysis process is necessary to identify the essence of the problem, separate from the client’s view on the problem. In the ideal case the client’s view may represent directly the problem of concern, though, in practice this is far from truth and additional steps are required to capture the real problems.

The technical problem analysis process is related to the concepts Technical Problem and Sub-Problem and the functions Select, Search and Discover in the model of Figure 1. It consists of the following steps:

1. Generalizing the requirements: whereby the requirements are abstracted and generalized.
2. Identification of the sub-problems: whereby technical problems are identified from the generalized requirements.
3. Specification of the sub-problems: whereby the overall technical problem is decomposed into sub-problems.
4. Prioritization of the sub-problems: whereby the identified technical problems are prioritized before they are processed.

In the following we explain these processes in more detail.

4.2.1 Generalizing 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 interpretations 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

\(^4\) In mathematics, solving a concrete problem by first solving a more general problem is termed as the Inventor’s Paradox [44] [34]. The paradox refers to the fact that a general problem has paradoxically a simpler solution than the concrete problem.
entirely different terms and therefore may suggest and help to discover problems that were not thought of in the initial requirements.

4.2.2 Identification of the 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 [40]. We need thus first to discover and describe the problem. Therefore, in the generalized requirement specification we look for the important aspects that needs to be considered in the software architecture design [58]. 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.

4.2.3 Specification of the 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 consists of describing its name, its initial state and its goal.

4.2.4 Prioritization of the 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 handle 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 [1] 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 [55] 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. In parallel with our generalization of the requirements we were able to define the different sub-problems, which are listed in the following:

P1. Provide transparent concurrency control.
   Goal: Determine the set of concurrency control techniques that are required and provide this in a reusable form.

P2. Provide transparent recovery techniques.
   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. Provide transparent transaction management techniques.
   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. Provide adaptable transaction protocols based on transaction, system and data criteria.
   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.

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 relates basically to the concepts Solution Domain Knowledge and Solution Abstraction and the functions Search and Extract in the model of Figure 1. The solution domain analysis process consists of the following activities:
1. Identification and prioritization of the solution domains for each sub-problem.
2. Identification and prioritization of the knowledge sources for each solution domain.
3. Extracting solution domain concepts from solution domain knowledge.
4. Structuring the solution domain concepts.
5. Refining the solution domain concepts.

In the following we will explain these steps in more detail.

4.3.1 Identification and prioritization of 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. Further, 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 [24]. 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 [48]. In [2], 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) [14]. 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.

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 [30]. In that case it is necessary to specify the solution domain.

4.3.2 Identification and prioritization of knowledge sources

Each identified solution domain may cover a wide range of solution domain knowledge sources that represent the content of the related knowledge. 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 relevance. 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 relevance quality factor refers to the relevance of the solution domain knowledge for solving the identified technical problem.

The relevance 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 [2]:

\[ \text{priority}(s) = f(\text{objectivity}(s), \text{relevance}(s)) \]

Hereby \( \text{priority}(s) \), \( f() \), \( \text{objectivity}(s) \) and \( \text{relevance}(s) \) 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 relevance 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.

<table>
<thead>
<tr>
<th>SUB-PROBLEM</th>
<th>SOLUTION DOMAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Transaction Management</td>
</tr>
<tr>
<td>P2</td>
<td>Concurrency Control</td>
</tr>
<tr>
<td>P3</td>
<td>Recovery</td>
</tr>
<tr>
<td>P4</td>
<td>Adaptability</td>
</tr>
</tbody>
</table>

The prioritization of these solution domains was defined in the above order from P1 to P4.

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, literature on transactions, and documentation on the existing car dealer 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.
Table 2: A selected set of the identified knowledge sources for the overall solution domain

<table>
<thead>
<tr>
<th>ID</th>
<th>Knowledge Source</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS2</td>
<td>Atomic Transactions [36]</td>
<td>textbook</td>
</tr>
<tr>
<td>KS3</td>
<td>An Introduction to Database Systems [13]</td>
<td>textbook</td>
</tr>
<tr>
<td>KS5</td>
<td>The design and implementation of a distributed transaction system based on atomic data types [68]</td>
<td>journal paper</td>
</tr>
<tr>
<td>KS6</td>
<td>Transaction processing: concepts and techniques [26]</td>
<td>textbook</td>
</tr>
<tr>
<td>KS8</td>
<td>Transactions and Consistency in Distributed Database Systems [61]</td>
<td>journal paper</td>
</tr>
</tbody>
</table>

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 specific aspects 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 [47][22][62]. In addition we organized a workshop on Adaptability in Object-Oriented Software Development [57][4].

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Knowledge Source</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS1</td>
<td>Concurrency Control in Advanced Database Applications [7]</td>
<td>journal paper</td>
</tr>
<tr>
<td>KS2</td>
<td>Concurrency Control in Distributed Database Systems [16]</td>
<td>textbook</td>
</tr>
<tr>
<td>KS3</td>
<td>The theory of Database Concurrency Control [41].</td>
<td>textbook</td>
</tr>
<tr>
<td>KS5</td>
<td>Concurrency Control and Reliability in Distributed Systems [12]</td>
<td>journal paper</td>
</tr>
<tr>
<td>KS6</td>
<td>Concurrency Control in Distributed Database Systems [9]</td>
<td>textbook</td>
</tr>
</tbody>
</table>

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 also provides detailed information on the solution domain Concurrency Control.

4.3.3 Extracting 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 [43], [25], [66].

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 [66], [39], [19], [21].

The concept formation process utilizes and abstracts from the collected knowledge to form concepts\(^{5}\). In the literature, several concept formation

\(^{5}\) 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 [23].
techniques have been identified\textsuperscript{6} [26][46][23]. One of the basic abstraction
techniques in forming concepts is by identifying the variations and
commonalities of extracted information from the knowledge sources
[52][20]. 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 assigned 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 [35]: \textit{Atomic Transaction System}, \textit{Transaction},
\textit{TransactionManager}, \textit{PolicyManager}, \textit{Scheduler}, \textit{RecoveryManager},
\textit{DataManager}, \textit{Data Object}.

4.3.4 Structuring the Solution Domain Concepts

The identified solution domain concepts are structured using
generalization-specialization relations and part-whole 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 represent concepts
and the edges between the nodes represent conceptual relations. The
notation of concept graphs is given in Figure 2.

The notation for a concept is a stereotype of the class notation in the
Unified Modeling Language [13]. A stereotype represents a subclass of a
modeling element with the same form but with a different intent. The
stereotype for a concept is identified by the keyword \texttt{<concept>}\textsuperscript{7}.

\textsuperscript{6} This process of concept abstraction is usually considered as a psychological activity that is
often associated with the term 'experience' [52]. 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.

\textsuperscript{7} Note that a class does not need to 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. Although the notation that we use for representing concepts is similar to the
notation of classes, one should be aware that concepts are at a different abstraction level
than classes and should be treated as such.
EXAMPLE

Figure 3 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. This means that Transaction Manager is responsible for the atomic execution of Transaction. The association relation manages between concept DataManager and Data Object represents the maintenance of the consistency of data objects. Hereby, DataManager utilizes and coordinates the concepts Scheduler and RecoveryManager by means of the association relation coordinates. The concept PolicyManager coordinates the activities of the concepts TransactionManager and DataManager and defines the policy for adapting to different transaction protocols. Finally, the association relation accesses between Transaction and Data Object defines a read/update relation between these two. A more detailed description of these concepts is given in [35].
4.3.5 Refinement of Solution Domain Concepts

After identifying the top-level conceptual architecture we focus on each sub-problem and essentially follow the same synthesis process. The refinement becomes 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. Due to space limitations we will not elaborate on the refinement of these concepts in this chapter but suffice to refer to [35] in which this is described in detail.

4.4 Alternative Design 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 Defining the Alternatives for each Concept and Describing the Constraints. Let us now explain these sub-processes in more detail.

4.4.1 Defining the Alternatives for each Concept

In Synbad, the various architecture design alternatives are largely dealt with by deriving architectural abstractions from well-established concepts in the solution domain. 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.

The alternatives of a given concept may be explicitly identified and published. 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 then be too large to provide a name for each of them individually. Nevertheless, we need to depict the total set of alternatives so that each of them can be derived if necessary. For this, first
the alternatives of each sub-concept are identified, and consequently the various compositions of these alternatives are considered.

**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*. The table 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*.

<table>
<thead>
<tr>
<th>A. SYNCHRONIZATION SCHEME</th>
<th>B. SYNCHRONIZATION STRATEGY</th>
<th>C. PERFORMANCE FAILURE DETECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Two Phase Locking</td>
<td>Aggressive</td>
<td>Deadlock Detector</td>
</tr>
<tr>
<td>2. Timestamp Ordering</td>
<td>Conservative</td>
<td>Infinite Blocking Detector</td>
</tr>
<tr>
<td>3. Optimistic Serial</td>
<td></td>
<td>Infinite Restart Detector</td>
</tr>
<tr>
<td>4. Serial</td>
<td></td>
<td>Cyclic Restart Detector</td>
</tr>
</tbody>
</table>

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 conservative deadlock detection mechanism is used. Note that the concept *Scheduler* has $4 \times 2 \times 4 = 16$ theoretically possible alternatives.

Another example is given in Table 5, which represents the alternative space for the concept *RecoveryManager*.

---

*These have been derived in the refinement process in which the synthesis process has been applied to define the sub-architecture of the concept *Scheduler* [35].
Table 5: Alternatives of the sub-concepts of RecoveryManager

<table>
<thead>
<tr>
<th>A. LOG MANAGER</th>
<th>B. FAILURE ATOMICITY SYNCHRONIZER</th>
<th>C. RESTARTING</th>
<th>D. CHECKPOINTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Operation Logging</td>
<td>Recoverable</td>
<td>Undo / Redo</td>
<td>Commit-Consistent</td>
</tr>
<tr>
<td>2. Deferred-Update</td>
<td>Cascadeless</td>
<td>No-Undo / Redo</td>
<td>Cache-Consistent</td>
</tr>
<tr>
<td>3. Update-In-Place</td>
<td>Subject</td>
<td>Undo / No-Redo</td>
<td>Fuzzy</td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>No-undo / No-redo</td>
<td></td>
</tr>
</tbody>
</table>

An alternative of the concept RecoveryManager is the tuple (Operation Logging, Strict, Undo-Redo, Commit-consistent), representing a RecoveryManager that applies Operation Logging, Strict executions, adopts Undo-Redo algorithm in case of restarts and a Commit-Consistent checkpointing mechanism for optimizing the restart procedure. The total number of theoretically possible alternatives of RecoveryManager is 4x3x4x3=144. If alternatives of Scheduler and RecoveryManager are composed then the number of the set of possible alternative compositions equals 16x144=2304 alternatives. Obviously, not all the alternative compositions are possible or required and it is worthwhile to eliminate these alternatives. This process is described in the following section.

4.4.2 Describing Constraints between Alternatives

An architecture consists of a set of concepts that together define a certain structure. An instantiation of an architecture is a composition of instantiations of concepts [2][35]. 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 first describe the constraints among the sub-concepts within a concept and later among the peer-concepts. We use the Object Constraint Language (OCL) \cite{64} 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 possibly indicate a wrong decomposition. This may result in a reconsideration of the identified architectural structure of each concept.

4.5 Architecture Specification


4.5.1 Extracting Semantics of the Architecture

To provide a more formal specification the semantics of each individual concept is extracted from the solution domain. As a format for writing a formal specification we use:

\[ <\text{operation}><\text{pre-condition}><\text{post-condition}> \]

Hereby, \text{operation} represents the name of the operation of a concept. The part \text{pre-condition} describes the conditions and assumptions made about the values of the concept variables at the beginning of \text{operation}. The part \text{post-condition} describe what should be true about the values of the variables upon termination of \text{operation}. Note that this is just one particular way of specifying architectures. For the specification of transaction architectures this type of specification was appropriate, however, other applications may require different specification mechanisms.

4.5.2 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 the so-called collaboration diagrams are utilized \cite{13}. Collaboration diagrams show the structural organization of the components and the interaction among these components. The collaboration diagrams are derived from the pre-defined specifications of the architectural concepts.
EXAMPLE

The collaboration diagram for the transaction architecture is given in Figure 4.

![Collaboration diagram for the atomic transaction architecture]

Figure 4: Collaboration diagram for the atomic transaction architecture

5. DISCUSSION AND CONCLUSIONS

In this chapter we have presented Synbad, the synthesis-based software architecture design approach. This approach is based on the concept of 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. The novelty of Synbad 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 to the technical problems providing a more objective and reliable description of the problem. During the solution domain analysis, stable architectural

---

* The web site of this approach is at: http://trese.cs.utwente.nl/architecture_design/
components with rich semantics are derived from the solution domain knowledge that includes well-defined and stable concepts. The solution domain analysis itself is leveraged by the pre-identified technical problems so that the right detail of the solution domain model is guaranteed. 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 for the design of an atomic transaction architecture for a distributed car dealer system in an industrial project. 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 [63], 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, though, also applied on domains that are less formalized. In [5], for example, a software architecture has been derived for a Quality Management Systems for efficient information retrieval and in [67] 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 Synbad 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 4.1 and 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 the adaptability requirement of transaction protocols was interpreted only in a limited sense and referred to the adaptation of 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 have directed and scoped 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 excluded from 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 helpful and necessary indications on 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 4.3, 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 helpful.

2. Solution domain provides stable architectural abstractions

Synbad provides an explicit solution domain analysis process for identifying the right abstractions. After the analysis of the solution domain on transaction theory it appeared that this 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 4.3 we have 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 level of detail.

The requirement of stable solution domains in Synbad 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.

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 4.5 and in [35] 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 [35].

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

In Synbad, alternative space analysis is an explicit process. Thereby, for each concept the set of alternatives is described and constraints are defined among these alternatives. This together results in a depiction of the set of possible alternative designs, that is, alternatives design space, that may be derived from the given software architecture. As described in section 4.4 we have, for instance, defined the alternatives for the concepts Scheduler and RecoveryManager. From the solution domain analysis we have extracted the constraints within each of these concepts and constraints that apply among alternatives of these concepts [35]. We had two problems in the alternative space analysis process for the example project. First, although we had derived the conceptual architectures from the solution domain itself, during the alternative design 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 identifying these constraints is very
time-consuming. Defining constraints of solution domain concepts requires an improved 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 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.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for Scientific Research (NWO, ‘Inconsistency management in the requirements analysis phase’ project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

6. REFERENCES

56. B. Tekinerdoğan. Reliability problems and issues in a distributed car dealer information system, INEDIS project, 1996.


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