Designing Software Architectures as a Composition of Specializations of Knowledge Domains

Mehmet Aksit[1], Francesco Marcelloni[2],
Bedir Tekinerdogan[1], Charles Vuijst[1] and Lodewijk Bergmans[1, 3]

[1]TRESE Project, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {aksit | bergmans | bedir }@cs.utwente.nl
www server: http://wwwtrese.cs.utwente.nl
[2]Department of Information Engineering, University of Pisa,
Via Diotisalvi, 2-56126, Pisa, Italy.
email: france@iet.unipi.it
[3] STEX bv, D.Dijkhuisstraat 248, 7558 GG, Hengelo, The Netherlands

Abstract

This paper summarizes our experimental research and software development activities in designing robust, adaptable and reusable software architectures. Several years ago, based on our previous experiences in object-oriented software development, we made the following assumption: 'A software architecture should be a composition of specializations of knowledge domains'. To verify this assumption we carried out three pilot projects. In addition to the application of some popular domain analysis techniques such as *use cases*, we identified the invariant compositional structures of the software architectures and the related knowledge domains. Knowledge domains define the boundaries of the adaptability and reusability capabilities of software systems. Next, knowledge domains were mapped to object-oriented concepts. We experienced that some aspects of knowledge could not be directly modeled in terms of object-oriented concepts. In this paper we describe our approach, the pilot projects, the experienced problems and the adopted solutions for realizing the software architectures. We conclude the paper with the lessons that we learned from this experience.

Paper category: Experience paper

Keywords: Software architecture, software engineering practices

Correspondence address: Mehmet Aksit, TRESE Project, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

1. Introduction

Object-oriented methods aim at providing natural ways for decomposing (or composing) a system into (from) objects that correspond to concepts in the customer's problem domain[1]. For example, in OMT [Rumbaugh 91] nouns in the requirement specification, which are assumed to represent concepts, are considered as candidate objects. The identified objects are the basic building blocks of the object-oriented system to be constructed. In order not to disregard relevant objects, most methods advise software engineers to take dedicated steps such as reading books about the problem domain, interviewing customers, etc. [Coad 91]. The method Object-Oriented Software Engineering [Jacobson 92] proposes *use cases* as a systematic means to understand the problem domain.

We consider two important concerns in understanding the problem domain. First, it is very important to identify all the objects that are required for defining a consistent system, at least in its minimum configuration. Second, identified objects must serve as composable building blocks to construct robust, adaptable and reusable *architectures*. In this paper, we use the term architecture to designate the gross structure of a software system represented as a high level organization of computational elements and interactions between those elements as defined in [Garlan 95]. It has been shown in [Shaw 95] that the choice of architectural style can have far-reaching consequences because they shape the analysis of the problem and the expression of the design. Within the object-oriented community, issues related to domain analysis have been extensively discussed in various papers [Prieto-Diaz 91], workshops [OOPSLA_Workshop 95] and panels [OOPSLA_Panel 95].

For several years, we have been carrying out research activities to define guidelines for deriving object-oriented models for our customer's problem domains. In particular, we wanted to find answers to the following questions: First, would it be possible to define rules for identifying an 'ideal' software architecture? The absence of a good architecture generally predicts difficulties in realizing a software system in a robust, adaptable and reusable way. Second, what would be the obstacles that one might experience in mapping domain information into object-oriented models? Third, what kind of research activities would be needed to address the identified problems, if any. Last but definitely not least, would experiences gained from the pilot projects conflict with the theoretical assumptions? This paper presents our findings in this experimental research.

Based on previous experiences in object-oriented software development [Aksit 92b], we started with the following assumption: "A software architecture can be defined as a composition of specializations of knowledge domains". The compositional structure must reflect the invariant compositions of the problem domain. For example, the generic architecture of a simple motorized vehicle results from the composition of the knowledge about engine, chassis, and steering and breaking systems.

To verify this assumption, we carried out 3 pilot projects, and implemented and tested them extensively. We tried to define architectures based on the 'ideal architecture' concept. At this stage we experienced three problems. First, we spent a considerable amount of time in searching and understanding the related knowledge domains. Nevertheless, in all the pilot projects, we could extract satisfactory information from the literature. Second, sometimes it was necessary to extend domain knowledge to make it suitable for architecture definition. Third, we realized that mapping knowledge into object-oriented concepts was sometimes difficult, because certain aspects of knowledge could not be represented directly in terms of object-oriented concepts.

The paper is organized as follows: The following section summarizes our objectives in carrying out this experimental research. Section 3 outlines the method used in our pilot projects. Section 4 describes the initial requirements for the pilot projects. Section 5 explains how the architectures of frameworks are identified. Section 6 describes the realization of the frameworks. Experienced problems in mapping descriptions of architectures to object-oriented models are explained in section

1 The customer represents the person(s) who is interested in the solution of a software development problem.

6.1. Section 6.2 gives information about the implementations of the frameworks. Section 7 presents the lessons learned and research issues. Finally, section 8 concludes this paper. The appendix gives a selected presentation of the developed object models.

2. The Objectives

From 1987 to 1993, we have been involved in the development of a large number of pilot applications using, what we considered, the best of the available methods [Aksit 92b]. One of the conclusions was that architectural definitions of software systems are crucial in achieving a high degree of robustness, adaptability and reusability. In 1993, we decided to continue with our experimental activity in the direction of identifying and specifying software architectures. The intention of this work was to achieve the following three objectives:

- To define an approach towards the definition of object-oriented software architectures, with an emphasis on robustness, adaptability and reusability;
- To test this approach for realistic problems to see whether the approach met our expectations;
- From these practical experiences, to identify obstacles that may be experienced in realizing software architectures using commercially available object-oriented methods and languages. This serves two goals: firstly, to make software engineers aware of the potential pitfalls. Secondly, to provide an input to research activities.

3. The Approach

3.1 The Architecture Concept

Figure 1 illustrates our architecture concept. This architecture consists of 4 components which are considered necessary in providing the expected behavior of the software system. As an example, these components may represent the engine, chassis, breaking and steering systems of a simple motorized vehicle.

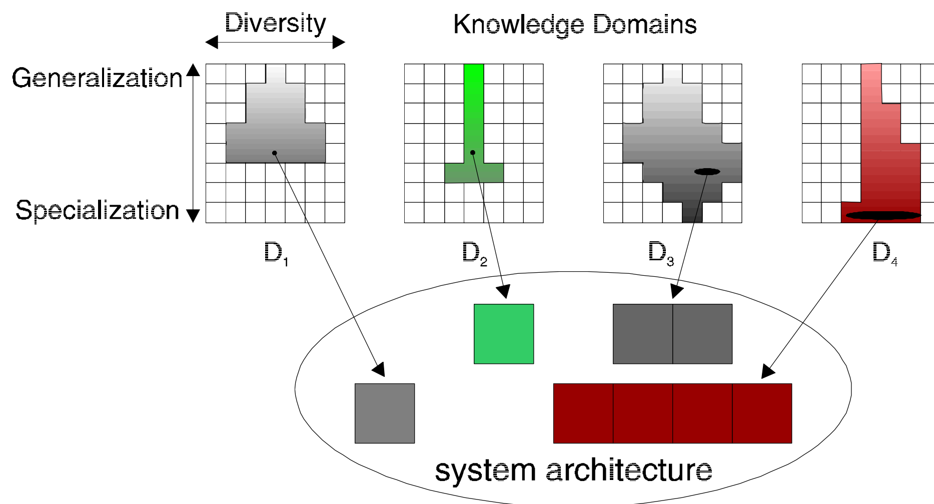

Figure 1. An example architecture as a composition of specialization of knowledge domains D1 to D4.

The architecture is a particular composition of specializations from the related knowledge domains D_1 to D_4. Each knowledge domain is modeled as a matrix. Here, rows and columns represent generalization/specialization and diversity relations among matrix elements, respectively. Each element in a matrix represents a concept in the corresponding knowledge domain. The granularity of matrix elements, rows and columns must be meaningful and consistent with respect to the available knowledge. The top row represents a knowledge domain in its most general form. Each row –except

the top row– is a specialization with respect to its higher adjacent row. There is no particular order among the elements in a row. The shaded areas show the relevant knowledge in each domain. The exact links between elements are not shown here because it is considered irrelevant for the purpose of this illustration.

This approach to representing knowledge fits in with the human way of thinking and reasoning. In the area of knowledge representation and expert systems, the techniques of *frames* [Minsky 75] and *semantic networks* [Levesque 79] can be considered as formal models that can be used to represent the same kind of knowledge organization. We consider our aim to be different however, in two ways: Firstly, our intention is to gather and structure informal knowledge during domain analysis. Secondly, we do not (yet) have intentions to prove hypotheses, as is the case for expert systems.

Defining architectures in this way has at least 3 advantages. First, it ensures that architectures are based on stable structures. The top-level decomposition of an architecture reflects the invariant concepts of that application. Each realization of these concepts is a specialization of existing knowledge. The higher-level concepts in a given knowledge domain generally correspond to theories. Most changes appear as a diversification and specialization of existing knowledge; theories hardly ever change. For example, although there have been considerable achievements in car industry, the minimum compositional architecture of motorized vehicles hardly changed during the last 50 years. The theories on combustion engines and physical laws are still valid. During the years, what we have experienced is the specialization and diversification of knowledge in building vehicles.

Second, such architectures are highly adaptable. The shaded area in a knowledge domain indicates the adaptability space; the architecture must be composed from these concepts. This area is open-ended since new specializations can be added at the bottom of the hierarchy.

The adaptability space for instantiations of the architecture is restricted in two ways however:

- To be selected as an architectural component, a concept must be realizable. It should not be too abstract for building software components. For example, the concept *Combustion Engine* in its most general form can be too abstract to include in building vehicles.
- Concepts from different domains may restrict each other. For example, a chassis may not be strong enough to carry very powerful engines.

Last, such architectures can be highly reusable. Each component is derived from existing knowledge. If knowledge domains can be mapped to software architectures effectively, then the software system will be as reusable as the available knowledge. One may not expect software engineers to design software architectures any better than their understanding of the concepts within the theory itself. The rationale for this statement is that the availability of a certain amount of domain knowledge and theory is a strong indication that this knowledge is stable and applicable in many circumstances.

We should note that, although we consider these knowledge domains as *existing* knowledge, in a lot of cases the software engineer will actually introduce new specializations at the bottom that have not been described elsewhere before. This represents knowledge that is specific to a particular application or customer situation.

3.2 The Process

In our pilot projects, to identify architectures, we adopted the process shown in Figure 2. This process has 5 steps[2]. First we identify use cases [Jacobson 92] based on the problem description. We experienced that this provides a better understanding of the requirements.

---

2 For simplicity, we omitted details such as interactions with the customer and iteration paths.

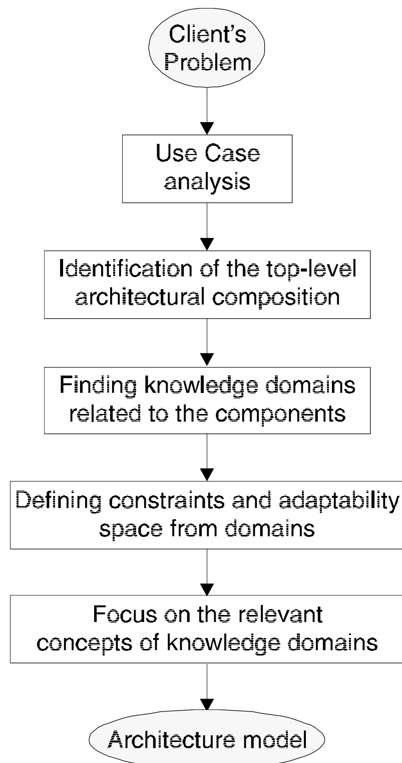

Figure 2. A process for architecture definition.

Second, after use case analysis, we identify the top-level conceptual architecture. This requires searching the related literature and finding similarities among various publications. We also try to discover concepts that are indispensable for a given problem. For example, we search for the minimum configuration by gradually excluding concepts until essential characteristics of the application are lost.

Third, for each component in a composition, we search for related knowledge. For each knowledge domain, we organize the gathered information similar to the matrix structure, as shown by Figure 1. The matrix elements are represented as a piece of text or as mathematical formulas. The generalization/specialization and diversity discrimination is realized by systematically comparing and ordering concepts.

Fourth, for each domain, we identify which elements in a matrix can be included into the application being considered. A set of semantically correct alternatives depict here the adaptability space. In addition, we investigate whether specializations from different domains enforce constraints on each other, if they are included within the same architecture. Additional user-defined constraints can be added, for example to restrict the scope of the architecture.

The last step is to focus on the relevant portion of the knowledge domains, since possibly not all the conceptual elements in the specialization hierarchy are relevant for a given problem. We define a path from the selected concepts to the concepts defined at the highest abstraction level. All the concepts along this path are necessary in realizing the selected specializations.

The next step is to map the architecture model into an object-oriented model. During this phase, whenever we have difficulties in effectively mapping architectural descriptions into object-oriented models, we examine object-oriented methods and design patterns [Gamma 95] to understand how these problems are addressed by them. In cases where we cannot find a solution to our problem, we refer to related research work.

4. Description of the Pilot Projects

In the following paragraphs we will describe the initial requirements for the pilot projects.

Transaction Framework

Our first pilot project aims at designing an object-oriented atomic transaction framework to be used in a distributed car dealer management system[3]. Data and processing in a car dealer management system are largely distributed and therefore *serializability* and *recoverability* of executions are required. Using atomic transactions [Bernstein 87], serializability and recoverability for a group of statements can be ensured. Serializability means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same set of 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.

A car dealer management system may be constituted of a large number of applications with various characteristics, operates in heterogeneous environments, and may incorporate different data formats.

3 This project is carried out together with Siemens-Nixdorf Software Center and supported by Dutch Ministry of Economical affairs under the SENTER program.

To achieve optimal behavior, each of these aspects may require a transaction system with a dedicated serialization and recovery techniques. This requires transactions with dynamic adaptation of transaction semantics, optimized with respect to the application and environmental conditions and data formats. The adaptation policy therefore, must be determined by programmers, the operating system or by the data objects.

A car dealer management system is large, complex and long-lived. Reusability of software is therefore considered as an important requirement to reduce maintenance costs.

Image Processing Framework
At the laboratory for Clinical and Experimental Image Processing, located at the university hospital Leiden, an image processing system is being developed for the analysis of the human heart [Zwet 94]. Traditionally, image processing algorithms have been implemented at the laboratory using procedures. For example, assume that the application of three image processing algorithms $algorithm_1$, $algorithm_2$ and $algorithm_3$ on the input image would produce the output image:

```
outputImage = algorithm3 (algorithm2 (algorithm1 (inputImage)));
```

The output parameter of the first algorithm is the input parameter of the second algorithm and the output parameter of the second algorithm is the input parameter of the third algorithm. Here, all cascaded input-output parameters of these algorithms must be compatible. Procedures are largely dependent on the representation of the input and output parameters [Wegner 84].

In object-oriented modeling, algorithms can be defined as operations of a class *Image* and the structure of *Image* can be encapsulated within its private part. By sending cascaded messages one can transform images subsequently:

```
outputImage = ((inputImage.algorithm1).algorithm2).algorithm3;
```

Here, *inputImage* receives the message $algorithm_1$ which results in a new image that receives the message $algorithm_2$, and so on. Provided that each image understands these messages, one may apply the algorithms to images in any order.

The object-oriented approach looks promising because image formats can be encapsulated and abstracted by image processing algorithms. This means, however, that each image must define all the required image processing algorithms, which may demand a large number of method definitions. The second concern is to define an object-oriented image processing framework which is expressive enough to construct virtually any image processing algorithm that can be used for medical imaging. Last, effective code reuse can simplify implementation of image processing algorithms and decrease the maintenance costs.

Fuzzy-Logic Reasoning Framework
For several years, we have been carrying out research activities in formalizing the object-oriented software development process [Aksit 96]. One of the problems in modeling a software development process is to represent design uncertainties. As a result of our research, we concluded that fuzzy-logic theory [Dubois 80] might be useful for this purpose. For the practical implementation of our ideas, we decided to build a fuzzy reasoning framework [Marcelloni 95a].

A fuzzy reasoning system is characterized by two basic features. First, a fuzzy reasoning process has the ability of deducing a possibly imprecise but meaningful conclusion from a collection of fuzzy rules and a partially true fact. Second, such a reasoning process is executed by using rules and facts codified in a natural language.

Consider, for example, the following rule: “If an entity is relevant within the problem domain then select it as a class”. Two-valued logic forces the software engineer to take abrupt decisions, such as, “the entity is relevant” or “not relevant”. Therefore, expressing uncertain information using two-valued logic can be quite tedious. A fuzzy reasoning system, however, can accept input values such as “the entity is weakly relevant”. A fuzzy-rule can be expressed using linguistic expressions. For example, the fuzzy-rule “If an entity is weakly relevant then it is weakly likely to correspond to a class” can reason about the entities that are qualified as *weakly relevant*.

The design of a fuzzy-logic reasoning framework for our purpose requires a number of considerations. First, fuzzy-logic may be based on different implication operators. Second, in fuzzy reasoning, the semantics of the connectives AND and ALSO can be interpreted in various ways. Third, the framework must also process linguistic values, such as *weakly relevant*, instead of the Boolean variables *true* and *false* only.

In addition to fuzzy-logic specific requirements, we think that the framework must provide both goal-driven and data-driven activation modalities. Since contextual information plays a significant role in a software development process, the rules must be dynamically adapted to the changing context. Finally, the framework must be able to execute two-valued logic based reasoning as well.

Comparison of the Pilot Projects

Table 1 summarizes the initial requirements for the pilot projects. All the projects aim at defining object-oriented frameworks [Johnson 88] rather than developing a dedicated software system for a given problem. The required features of these frameworks are quite different because they relate to different application domains. The key requirements of these frameworks, however, are quite similar. They all must support different kinds of implementations. For example, the transaction framework must provide different serialization techniques, the image processing framework must be able to express any image processing algorithm and the fuzzy-logic reasoning framework must be able to implement different implication rules. In addition, for all frameworks adaptability and reusability are major requirements.

Pilot project	Features	Key requirements	Application area
transaction framework	serializability, recoverability	programmer/system/object defined policies, dynamic adaptability, reuse	car dealer management system
image processing framework	many possible algorithms and representations	no restrictions on image representations and algorithms, reuse	medical imaging system
fuzzy-logic reasoning framework	different implementations of generalized modus ponens, easy definition of linguistic variables	support different implications, different implementations of the connectives AND and ALSO, goal-driven and data-driven activation, dynamic adaptation to context, both deterministic and fuzzy reasoning, reuse	representing uncertainty in object-oriented methods and CASE environments

Table 1. Summary of the requirements.

5. Architecture Definition

5.1 Identification of the Top-Level Decomposition of Architectures

Transaction Framework

A considerable number of text books and articles have been written on atomic transactions [Bernstein 87] [Moss 85]. After searching and comparing the literature, we noticed that most publications adopt a similar architecture. Figure 3 shows a representative architecture for illustrating the essential components of transaction systems.

The arrows in the figure indicate the interaction patterns between the components. The component *Transaction* represents a transaction block as defined by the programmer. The *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.

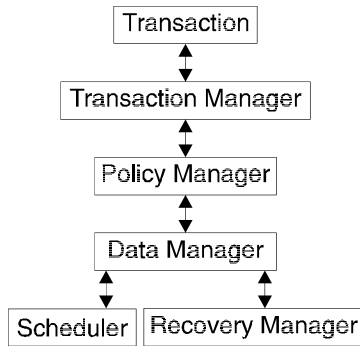

Figure 3. Essential components of a transaction system.

The *PolicyManager* determines the strategies for adaptation to different transaction semantics. 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 component. The component *DataManager* controls the access to its object and includes the components *Scheduler* and *RecoveryManager*. The component *Scheduler* orders the incoming messages to its object to achieve serializability. *Scheduler* may include deadlock avoidance and/or detection mechanisms. The component *RecoveryManager* keeps track of changes to its object to recover from failures.

Image Processing Framework

As stated in section 4, the architecture of the image processing system must be capable of expressing virtually any image processing algorithm suitable for medical imaging. Therefore, we had to search for techniques which could cover the area of image processing. After a thorough literature survey, we came across the theory of *image algebra* which is capable of expressing *almost all* the image-to-image transformations [Ritter 87a, 87b, 90]. The decomposition of the image processing framework is derived from this theory as depicted in Figure 4.

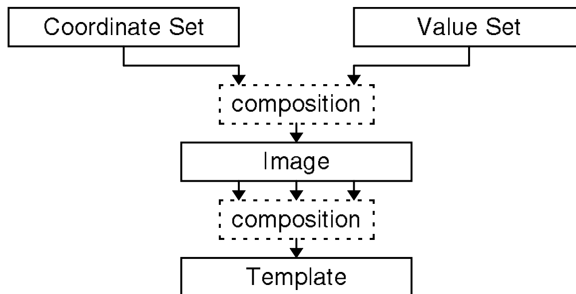

Figure 4. Decomposition of the image processing framework.

The image processing framework consists of *coordinate* and *value* sets. Images can be expressed as a composition of these two sets. In the theory of image algebra, the concept of image templates is introduced. An image processing algorithm, in general, can be defined as

```
anOutputImage =
anInputImage.anAlgebraicOp(aTemplate)
```

Here, anOutputImage represents the resulting image, *anInputImage* is the image to be processed, *anAlgebraicOp* is one of the basic operations defined by image algebra, and the argument *aTemplate* represents the algorithm to be applied on *anInputImage*. If templates can be generated from requirements specifications easily, this approach overcomes the problem of defining a large number of operations for each image, as only a few algebraic operations are required.

Fuzzy-Logic Reasoning Framework

A large amount of publications have been written on fuzzy-logic reasoning (for example [Lee 90], [Turksen 93], [Dubois 80], [Zimmermann 91]). After investigating the available literature, we concluded that the architecture shown in Figure 5 conforms to the concepts in most of these publications.

We selected the so-called *generalized modus ponens* (G.M.P.) as the basic inferencing technique because of its common usage in the literature. In the most general form, the generalized modus ponens may be expressed in the following way:

For a given rule R = “If A Then B”, and a fact A’, the conclusion B’ inferred by A’ is equal to A’ ∘ R, where ∘ is a *composition relation* between the fuzzy sets corresponding to A’ and R.

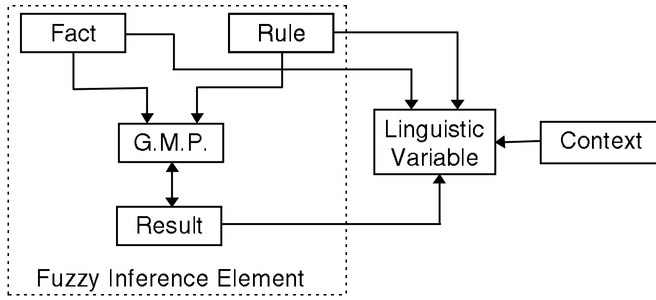

Figure 5. Decomposition of the fuzzy reasoning framework.

In this figure, the component *Fuzzy Inference Element* implements the inference mechanism. This element contains *Rule*, *Fact*, *G.M.P.* and *Result*. The components *Rule* and *Fact* represent the rules and facts as defined in generalized modus ponens. During the initialization phase, *Rule* and *Fact* communicate with the component *Linguistic Variable* to create a representation of themselves in terms of fuzzy sets. For each proposition involved in the *Rule* and *Fact*, the corresponding fuzzy set is created. The output values of *Rule* and *Fact*, again expressed in terms of fuzzy sets, are provided to the component *Generalized Modus Ponens* (*G.M.P.*). This component carries out the inference process and generates a result. The component *Result* combines all the outputs of the related generalized modus ponens components using the connective ALSO. The result of this combination is also expressed in terms of fuzzy-sets. The component *Linguistic Variable* is used to 'defuzzify' the fuzzy set produced by *Result*. In case of a goal-driven inference, the component *Linguistic Variable* asks from *Result* to provide the resulting value. In case of a data-driven inference, however, the request comes from the component *Result*. The defuzzification operation executed by the component *Linguistic Variable* converts the fuzzy set into a crisp value or approximates it as a linguistic value.

Particular to our framework is the component *Context*. As specified in the initial requirement specification, the validity of rules used in a software development process largely depends on changes in the context, and therefore, an explicit formulation of the effects of context is mandatory. The component *Context* is an instance of the whole fuzzy reasoning framework shown in Figure 5. *Context* reasons about the context information and may ask the component *Linguistic Variable* to modify the related semantic rule within the linguistic variable. Notice that the component *Context* may also include a sub-component *Context*, thereby allowing specification of the effects of the context on a context, etc. If the component *Context* is omitted, then the interpretation of linguistic values is fixed and cannot be changed dynamically.

5.2 Finding Knowledge Domains

Transaction Framework

To detail the transaction architecture shown in Figure 3, we investigated publications related to each component. We organized the available information for each component as a matrix structure.

The knowledge domain *TransactionManager* includes a variety of commit and abort protocols.

The knowledge domain *PolicyManager* is related to software/hardware performance, reliability modeling techniques [Carey 86][Agrawal 87] and decision making. Basically, application usage, transaction semantics and software/hardware architecture are the three major factors that determine the characteristics of a transaction. *PolicyManager*, therefore, may include models for these factors. Decisions can be made based on these models.

The knowledge domain *DataManager* is responsible for the coordination of *Scheduler* and *RecoveryManager*. *DataManager* deals with semantic interdependencies between these components. The knowledge domain *Scheduler* relates to scheduling, serialization and deadlock detection techniques. Finally, the knowledge domain *RecoveryManager* includes information from simple recovery algorithms to sophisticated stable storage recovery techniques.

Image Processing Framework

The architecture of our image processing framework is derived from the theory of image algebra.

The knowledge domains *Coordinate* and *Value Sets* are specializations of the set theory. These are homogeneous sets, meaning that all the set elements belong to the same type. By defining a small number of algebraic operations on sets as the primitive functions, one can conveniently construct different image processing algorithms. The knowledge domain is composed of coordinate and value sets because an image defines functional dependencies between these sets. Similarly, the knowledge domain *Template* defines functional dependencies among images and includes knowledge about image processing algorithms.

Fuzzy-Logic Reasoning Framework
For each component in Figure 5, we investigated the related knowledge domains. We summarize our findings in the following paragraphs:

The component *Linguistic Variable* represents a specialization of a language theory. A linguistic variable is characterized by a quintuple (x, T(x), U, G, M) where x is the name of the variable, T(x) is the term set of x, that is, the set of names of linguistic values of x with each value being a fuzzy set defined on U. Here, U is the universe of discourse, G is a syntactic rule for generating the names of values of x, and M is a semantic rule for associating with each value its meaning. A meaning is a fuzzy set defined in the universe of discourse of the linguistic variable. So, the knowledge domain *Linguistic Variable* must include knowledge about the definition of a (small) language with its syntax and semantics.

The component *Fuzzy Inference Element* reflects two theories: logic theory and fuzzy set theory. As explained in section 5.1, during a reasoning process, the components *Fact*, *Rule*, *Generalized Modus Ponens* and *Result* interact with each other. All these components adopt fuzzy sets as a common data structure to exchange information. The component Rule defines a rule. Here, the connective AND and the implication are implemented by a fuzzy conjunction and a fuzzy relation, respectively. The component *Generalized Modus Ponens* implements *the compositional rule of inference* as a *composition* between two relations. The component *Result* implements the aggregation operation which is an intersection or union between fuzzy sets. In the literature, many implementations of fuzzy conjunctions, implications, compositions, intersections and unions have been proposed.

Overview of the Related Knowledge
Table 2 shows the related knowledge domains of the pilot projects. It has been an extensive amount of work to find out the related knowledge domains from the literature. Nevertheless, for each domain, we could extract the information necessary to define stable architectures. For each component in an architecture, we could find plenty of useful information.

Pilot Project	Component	Related Knowledge Domains	References
transaction framework	scheduler	scheduling/serialization and deadlock detection/avoidance techniques	[Bernstein 87]
	recovery manager	recovery techniques	
	transaction manager	commit and abort protocols	
	policy manager	transaction/architecture/application performance and reliability models, decision making	
image processing framework	value and coordinate sets	set theory, mathematical domains, algebra	[Ritter 87a,b]
	images templates	function theory, image representation techniques, algebra, image processing	
fuzzy-logic reasoning framework	fuzzy inference elements	fuzzy set theory, logic theory	[Dubois 80] [Klir 88]
	linguistic variables	language theory, fuzzy set theory	[Zadeh 73]

Table 2. Summary of the related knowledge domains.

5.3 Defining Constraints and Adaptability Space

Transaction Framework

Obviously, the interaction protocols among the components of a transaction must be compatible. For example, the commit and abort protocols of *TransactionManager* must be understood by the corresponding *DataManagers*. If the protocols of *TransactionManager* are changed, then the protocols of the *DataManagers* must change accordingly. Similarly, adapting transaction semantics must be carefully managed. In section 4, one of the important requirements of our transaction framework was defined as a need to adapt transaction semantics determined by programmers, the operating system or by the data objects. If the transaction system is dynamically changed, for instance by the operating system, then the components *Serializer* and *RecoveryManager* must be adapted accordingly.

In addition to interaction compatibility requirements, there may be restrictions on the composability of components. For example, we found out that the components *Scheduler* and *Recovery* are in some cases dependent on each other [Weihl 89]. Therefore, it is not always possible to combine every concept from the knowledge domain *Scheduler* with any concept from the knowledge domain *RecoveryManager*. In addition, we identified that the different serialization protocols adopted by the scheduler components may be incompatible with each other [Guerraoui 94]. Therefore, special care has to be taken to enforce composability constraints.

Image Processing Framework

There are 2 important constraints for the elements of coordinate and value sets. First, the elements of a set must be of the same type. For example, a coordinate set must only contain coordinates of a specific dimension type such as the frequency domain. Similarly, a value set must only contain values of a given type such as Boolean values for black-and-white images. Second, there may be some ordering relations among the elements of sets. In particular, the elements of a coordinate set must be ordered with respect to the semantics of the coordinate set. For example, in a two-dimensional spatial representation, the adjacent coordinates correspond to the image samples that are also physically adjacent to each other.

There are some restrictions imposed by the algebraic operations. An algebraic operation between two images may only be performed if both images have exactly the same coordinate set. In addition, the types of elements of value sets must be compatible.

Templates represent the functional dependencies between original and resulting images. We found that it is possible to categorize these dependencies into 8 different types. Namely, there are 7 variant and 1 invariant types of templates. A variant property is defined on certain aspects of coordinate and value sets. For example, a template may particularly consider whether a coordinate is at the boundary of an image or not. The categorization of templates helped us in defining a method to construct different kinds of templates in a systematic way. In our approach, the software engineer is asked to provide the variant and invariant properties of the templates. This information is sufficient enough to build templates in a straightforward way [Vuijst 94].

Fuzzy-Logic Reasoning Framework

The components of the fuzzy-logic reasoning architecture, as defined by Figure 5, can be seen as specializations of some aspects of the fuzzy-set theory. Theoretically, we can select each combination of specializations for implementing the reasoning process. For instance, in the component *Rule*, we can interpret the connective AND and the implication as a fuzzy conjunction which uses the minimum, and as the Mandami's implication, respectively [Mandami 77]. The component *Generalized Modus Ponens* may be implemented by the max-min compositional rule of inference defined by Zadeh [Zadeh 73]. The component *Result* may implement the connective ALSO as a union among fuzzy sets which uses the maximum. Not all the possible combinations, however, can produce meaningful results from a logical view point [Turksen 93] [Marcelloni 95b]. This means that fuzzy set theory is constrained by the logic theory in the fuzzy logic domain.

Overview of the Constraints and the Adaptability Space

As illustrated by Table 3, all the three frameworks require interaction and composability constraints to guarantee correct behavior. These constraints are the boundaries of the adaptability space of each framework.

Pilot Project	Adaptability Space	Inter-component constraints
transaction framework	scheduling and recovery concepts	intra-data manager (scheduler and recovery) and inter-data managers
image processing framework	different coordinate and value types, a large possible number of templates in 8 categories	sets must be homogeneous, ordering of elements in sets, type compatibility restrictions imposed by algebraic operations, 8 categories of templates
fuzzy-logic reasoning framework	several implementations of fuzzy reasoning, language used in the rules	rule, generalized modus ponens and result are constrained each other by logical soundness, rules and facts constrained by the linguistic variable

Table 3. Adaptability space and inter-component constraints.

6. Realization of the Frameworks

6.1 Experienced Problems in Mapping Architectures into Object-Oriented Models

During mapping architectures to object-oriented frameworks, we experienced a number of problems because the architectural concepts could not be directly mapped to the object-oriented concepts. As a consequence, we were forced to represent some architectural concepts in the implementation of operations of objects instead of explicit representations. This reduces adaptability and reusability of programs. The following sections explain some significant problems that we experienced during the development of the frameworks.

6.1.1 Dynamically Changing Implementations

This means that the implementation of an object is not fixed but can change at object initialization or execution time. There are two basic reasons why an object may dynamically change its implementation: improving the implementation, or evolution of the behavior. Improving the implementation can be necessary for example, for improving the speed and space performance of the implementation of objects, dealing with heterogeneous systems, etc. Evolution of the behavior of an object can be necessary for objects that represent evolutionary concepts.

In all the pilot projects dynamically changing implementations are required. For example, in the transaction framework shown in Figure 3, the components *Scheduler* and *RecoveryManager* have to be adapted dynamically with respect to changing application or system conditions.

Most transaction systems are distributed and long-lived. During the life-cycle of a transaction system, new commit and abort protocols, serialization and recovery algorithms may be introduced to cope with the changing demands of applications and system architectures. Suspending the transaction system and recompiling it with the new improvements may not be always preferable.

In the image processing framework, dynamically changing implementations are required mainly for improving the speed and space performance of algorithms. For example, implementing a spatial image as a matrix may not be space efficient if the matrix is sparse. On the other hand, matrix representation can be very time efficient for certain algorithms since each image element can then be directly accessible.

As explained in section 5.3, the components of the fuzzy-logic reasoning framework can be implemented in many different ways. The choice of a particular implementation affects the results of the reasoning. Such a choice is generally determined by the type of application and the input values. Therefore, only at run-time it is possible to determine which implementation allows to deduce the desired results. For most fuzzy-logic reasoning systems, instantiation of implementations during

object creation time would be satisfactory. For reasoning systems with learning behavior, however, the implementation of concepts may change dynamically.

The Strategy pattern [Gamma 95] can be used to define objects with dynamically changing implementations. In the Strategy pattern, different implementations are termed as *strategies* and are represented as objects. An object with changing implementations is termed as *context* and it aggregates one or more of the concrete strategies.

Now let us assume that C_c is the *context class* that requires a dynamic implementation and therefore it encapsulates its *strategy* object O_s. Here, O_s implements the methods m_1 to m_n. C_c declares these methods at its interface, but redirects the requests for these methods to O_s by invoking the corresponding methods on O_s. For example, the method m_1 is implemented by C_c in the following way:

```
Cc::m1(arguments)
  return Os.m1(arguments);
```

Provided that all the *strategy* objects implement the methods m_1 to m_n, by assigning a new strategy object O_{new} to O_s, one can change the implementations of the *context* object.

$$O_s := O_{new};$$

Here, the implementation of the *context* object is changed to O_{new}. Notice that the *strategy* object O_s behaves like a superclass because all its methods are visible at the interface of the *context* class C_c. Changing the implementation is equivalent to changing the super class of the object.

There are, however, a number of problems with this approach. First of all, the *context* class C_c must declare all the methods m_1 to m_n explicitly. If n is large, then this can be a tedious and error-prone task (O_s could have many methods defined in its superclasses). Second, the Strategy pattern cannot be used for evolving systems. The precise set of methods and their arguments has to be fixed when class C_c is defined since C_c has to declare all the dynamically changing methods explicitly. Third, although the *strategy* object behaves like the superclass, it cannot polymorphically refer to the *context* object through *self* calls. This is similar to the *self-problem* as defined by Lieberman [Lieberman 86].

An alternative to the Strategy pattern is to use the delegation mechanism [Lieberman 86]. Delegation is a mechanism that allows objects to share behavior. If a *server* object cannot respond to a particular request of its client, then it delegates the request to one or more *designated* objects. If one of the designated objects can execute the request, then it executes it on behalf of the server object. If needed, the designated object can refer to the server object by calling on the pseudo variable *self*. The delegation mechanism is similar to inheritance; the designated object behaves like the superclass of the server. Delegation can express dynamic implementations if the *context* object delegates to its *strategy* object. In case of delegation, the *context* object delegates all the requests that it cannot respond to, thereby eliminating the need for declaring the dynamically changing methods explicitly. This also supports the evolution of the *context* object. In addition, the pseudo variable *self* is provided by the underlying delegation mechanism.

6.1.2 Difficulties in Expressing Knowledge Specializations Using Class Inheritance

In our approach, the related knowledge domains are identified and represented using generalization, specialization and diversification relations, as described before. This process is performed from the perspective of modeling knowledge domains and solutions. It appears that the generalization-specialization hierarchies from the knowledge domains cannot always be mapped directly to the object-oriented inheritance hierarchies.

The reason for this is that generally object-oriented inheritance semantics are defined as inheritance of methods and instance variables from one or more superclasses by one or more subclasses. A subclass may add new methods and instance variables, and override existing methods. These semantics cannot always represent complex generalization, specialization and diversification relations among knowledge domain concepts.

For example in the transaction framework, the choice for a particular policy as made by the *PolicyManager* is the result of the application of many different rules and constraints. In a generalization-specialization hierarchy of *PolicyManagers*, gradually more rules and constraints are added. Mapping this hierarchy to a class-inheritance structure is far from trivial[4].

In the fuzzy-logic reasoning framework, the language-based specifications of linguistic variables require a grammar specification for scanning and parsing. In the generalization-specification hierarchy of the knowledge domain, new linguistic variables are added in specialization classes. This corresponds to the extension of the grammar rules. It is not possible to map this grammar-based hierarchy directly onto a class-inheritance hierarchy.

6.1.3 Architectural Constraints

We previously discussed (e.g. in sections 3 and 5.3) that a number of constraints must be enforced upon an architectural model. For example, architectural components cannot be mixed arbitrarily. We consider the enforcement of such constraints as fully distinct and independent from the application behavior: it is essentially a meta-level issue.

As an example, we refer to discussion in section 5.3 about the transaction framework, where many different specializations are available for both the components *Scheduler* and *Recovery*. One of the attractions of separating the *Scheduler* and *Recovery* components is that these are largely orthogonal, which allows for choosing independent concrete specializations. However, in a number of cases, these choices are *not* orthogonal: adopting a particular type of *Scheduler* excludes certain types of *Recovery*. This implies that whenever the composition is changed, the consistency of the new composition must be checked. This verification may involve interactions with multiple objects, and the verification specification must be modular so that both verification and application classes can be adapted and reused separately.

The enforcement of constraints on composition is currently typically achieved through type-checking mechanisms: by specifying a particular type for each of the components, we can ensure that only specializations of that type will be used as components and therefore these will satisfy some basic constraints. However, we already indicated that this is not always sufficient; a more powerful type checking mechanism than subclassing or signatures would be needed because several components and complex rules can determine the type correctness of programs.

In the more general case of constraints, the main difficulty is that we want the constraint specifications to be modular, but at the same time the enforcement of constraints may be required at many different locations and circumstances, which imposes maintainability problems.

6.1.4 Other Difficulties

Apart from the three problems that were just described, we briefly mention two other relevant issues that we had to deal with in realizing the architectures. We refer to the first issue as the *multiple views* problem. In the transaction framework, for example, the application objects that are involved in a transaction are accessed in two distinct ways: the application-specific functionality is invoked by other application objects (the user view). But the data management functionality that is specific for the transaction framework, such as locking or recovery methods are available as well, but should only be used by the transaction framework (the system view). The enforcement of such distinct views, which is important for retaining consistency, cannot be expressed in a convenient way by the conventional object model. The multiple views problem has been addressed in more detail in [Aksit 92a,b].

The second issue has been named as the *sharing behavior with state* problem. This problem is encountered whenever the behavior that is shared by multiple objects is affected by a particular state

4 Note that it *is* usually *possible* to implement an object-oriented application that provides correspondence to a domain knowledge hierarchy. However, this may well require the creation of additional structures and interactions because a one-to-one mapping is impossible.

that is shared by those objects as well. Sharing of behavior is usually achieved by a code reuse mechanism such as inheritance. If a shared behavior is affected by a shared data, however, class inheritance may not adequately model this situation. This is because, classes do not provide a means for sharing the state of instance variables; the instances of a class can share behavior through the class inheritance mechanism, but they cannot share the state which is encapsulated under the shared behavior because each instance will have its own instantiation of its state. Using an external *server* object for retrieving the state information weakens encapsulation. In addition, the polymorphic variable *self* then refer to the server object but not to the object that provides the shared behavior (*the self problem* [Lieberman 86]). For a more detailed analysis of the problem, the reader can refer to [Aksit 92b].

For example in the transaction framework, the behavior of the *PolicyManager* is shared by all *TransactionManager* objects. The method *chooseScheduler* is implemented by *PolicyManager* and reused by *TransactionManager*. A *PolicyManager* object collects all kinds of relevant system parameters and stores them in its instance variables. Here, the shared method *chooseScheduler* is affected by the shared state *system parameters*. The delegation mechanism can be used to solve this issue, as has been described in more detail in [Aksit 92b].

6.1.5 Overview

Table 4 provides an overview of the pilot projects, showing where certain difficulties were encountered, with a brief description of the area.

Pilot project	Dynamic implemen-tations	Inheritance vs. knowledge Specializations	Constraints	Multiple views	Sharing behavior & state
transaction framework	scheduling, recovery	policy manager	data manager	user-system views	system parameters
image processing framework	alternative implemen-tations	no	value & coordinate sets	no	no
fuzzy-logic reasoning	fuzzy-logic concepts	linguistic variables	implications	linguistic variables	no

Table 4. Pilot projects versus problems.

6.2 Implementation of the Frameworks

Transaction Framework

The transaction framework is implemented using the Smalltalk language. To change the implementations of *Scheduler* and *RecoveryManager*, we built a delegation mechanism upon the Smalltalk language. To implement this mechanism, each delegated message is *reified* and represented as an object. In the literature, this concept is known as *message reflection* [Ferber 89]. By changing the attributes of a message object and re-activating it again, one can realize a delegation mechanism.

In the implementation, constraints on object interactions and compositions are defined in separate classes. To enforce a constraint, the messages that may violate the constraints of objects are reified and redirected to the constraint classes. After verifying the validity of message invocations, the messages are re-activated again. If the constraints are violated, an exception is raised.

The prototype is currently running on a single machine. To implement the framework we mapped each architectural concept to a class. The implementation consists of 43 classes. Each knowledge domain is represented by inheritance hierarchies. The framework consists of 3 major inheritance hierarchies. Table 5 gives the number of classes defined within each inheritance hierarchy. The column *Time Spent* consists of two parts. Here, the *design time* indicates the total time spent in defining the architecture and analyzing and designing the framework. The *implementation time* shows the time spent for coding and testing. Detailed information about the transaction framework can be found in [Tekinerdogan 94].

In the current prototype, class *TransactionManager* implements a single commit/abort protocol. Class *PolicyManager* adopts a simple policy management strategy. We are currently implementing different protocols and an expert-system based *PolicyManager*. In addition, the transaction system will be ported to a distributed system platform so that it can be used within the implementation of the car dealer management system.

Image Processing Framework
The image processing framework is implemented using the C++ language. Each architectural concept is mapped to a C++ class. Similar to the transaction framework, interaction and composability constraints are enforced by defining constraints as meta-level classes and by reifying and redirecting the messages that may violate the constraints to these classes.

Currently, classes *Coordinate* and *Value Sets*, *Image* and *Constraint* enforcing classes are fully implemented. As an example, we implemented three templates: a low-pass filter, a fourier transform and image histogram templates. We also defined a method to guide the software engineer in creating templates conveniently [Vuijst 94].

Fuzzy-Logic Reasoning Framework
The fuzzy reasoning framework has been implemented using the Smalltalk language.

In the framework, class *LinguisticVariable* has two major methods for the fuzzification and defuzzification process. Class *LinguisticVariable* is the root of the inheritance hierarchy in which each subclass implements a different defuzzification strategy. At the moment, we have implemented only the most common defuzzification strategies.

Rules are organized in an inheritance hierarchy as shown in the appendix. Class *Fact* is composed by one or more *Propositions*. *Proposition* and *Rule* inherit from class *FuzzySet* which aggregates class *MembershipFunction*. We defined eight types of membership functions. The component *Generalized-ModusPonens* is implemented as a method of class *Rule* as it can be considered as an operation executed by the rule when a fact is provided to the rule. Currently, we considered only the generalized modus ponens as a fuzzy reasoning mechanism. We are now investigating other possibilities such as the *syllogisms* proposed by Zadeh [Zadeh 85]. Further, we are implementing more defuzzification strategies and membership functions. Alternatives implementations of the generalized modus ponens which can be used with particular implications and fuzzy sets are being analyzed. Such implementations enable considerable performances improvements.

Pilot Project	Language	Inheritance & # of Classes	Time Spent	Reference
transaction framework	Smalltalk	serialization hierarchy: 13 dead-lock hierarchy: 7 recovery hierarchy: 9 other: 15	design = 5 impl. = 2 months	[Tekinerdogan 94]
image processing framework	C++	single inheritance hierarchy: 20	design = 6 impl. = 2 months	[Vuijst 94]
fuzzy-logic reasoning framework	Smalltalk	rule hierarchy: 8 linguistic variable hierarchy: 4 membership functions hier.: 8 linguistic value hierarchy: 10 other classes: 29	design = 6 impl. =1 months	[Marcelloni 95a]

Table 5. Implementation aspects of the frameworks.

Comparison of Implementations
All the three implementations are directly derived from their architectural specifications. In addition to adopting 'standard' object-oriented models and design patterns, delegation and message reflection techniques are implemented to make software more adaptable and reusable. In all pilot projects, the designers were not experienced in the corresponding domains. Therefore, they spent a considerable amount of their time in understanding the related knowledge domains.

7. Lessons Learned and Research Issues

Our findings are summarized in the following items:

- *For certain knowledge domains specific inheritance semantics are necessary*: The method and attribute inheritance mechanism as defined by most object-oriented models are not always able to model generalization/specialization relations of the knowledge domains. In this case, the extension of the object-oriented model with some dedicated 'specification inheritance' mechanism is required to solve this in a modular and maintainable way.
- *Delegation based object-model is needed*: As discussed in section 6.1.1, we found the delegation mechanism quite necessary in defining adaptable software systems. For example, delegation can help in improving several design patterns such as *Strategy* and *Bridge* because it supports evolution of interfaces. In addition, delegation techniques can help dealing with the 'shared behavior affected by shared state' problem (see section 6.1.4). We do not consider, however, delegation as an alternative to inheritance or class concepts. Both delegation, inheritance and class concepts can co-exist together.

  We found some difficulties in explaining our delegation-based models to the software engineers working at the organizations where the pilot projects were carried out. These engineers were knowledgeable about the most popular object-oriented methods and programming languages, but the concept of delegation was unknown to them.
- *Enforcing constraints is essential, but not fully supported yet*: To instantiate and manage a dynamically evolving application while preserving its robustness requires high-level mechanisms to enforce the semantical constraints of that application. Strongly typed languages aim at detecting semantic errors as early as possible. We experienced, however, that type checking mechanisms of current strongly-typed object-oriented languages are not sufficient; type checking rules, in general, fail in detecting the complex interaction and composability constraints of objects.
- *Reflection techniques are useful*: Message reflection techniques are useful in implementing the delegation concept. In addition, by using message reflection, it is possible to modularly separate but 'functionally' integrate constraining classes and 'application classes'. This is particularly useful in enforcing interaction and composability constraints on objects in a modular way. However, in our implemetation, adopting reflection techniques to enforce constraints only offered run-time verification. Compile-time reflection techniques could be useful in improving the performance of our implementations. Similar to the concept of delegation, reflective modeling techniques were unknown to the most practical software engineers that we worked with.
- *Further research is needed in object-composition techniques*: In an architectural description, knowledge domains may model different aspects such as real-time, synchronization, coordinated behavior, etc. It has been shown by a number of publications that although separation of concerns is an essential concept for improving robustness, adaptability and reusability, composing separated concerns such as real-time and synchronization is far from trivial [Nierstrasz 95][Mullet 95] [Aksit 92a, 93, 94] [Bergmans 95]. Since software architectures can be defined as compositions of specializations of knowledge domains, we think that research activities for enhancing the composability capabilities of object-oriented models can be of great help; highly composable object models would improve the adaptability and reusability factors of software architectures.
- *New object-oriented process and product metrics are needed*: Defining software architectures first and then instantiate frameworks and object-oriented programs was uncommon to the organizations that we were dealing with. We were asked, right from the beginning, to start implementing object-oriented programs. We spent a considerable amount of time in explaining the necessity of designing software architectures. We could not however, justify our claims by using software metrics. Although recently there have been a number of publications [Abreu 94][Chidamber 94] on object-oriented metrics, we could not find them directly usable for the following reasons. Firstly, most metrics were applicable for final object models only. We needed process and product metrics for every design rule that we applied. Secondly, we found using threshold values, like most

metrics propose, quite meaningless. It is, in general, very hard to define ideal threshold values. Thirdly, most metrics do not explicitly model the effect of context, which we experienced as an essential parameter.

- *Software artifacts must be recorded, related and integrated*: During the software development process, from domain analysis to coding, lots of information were generated, processed and different kinds of models were built. These, so-called *software artifacts*, were recorded in various formats, from informal textual information to executable object-oriented programming concepts. We found it extremely difficult to record, trace and relate all the artifacts, although we used object-oriented CASE environments, hypertext-like tools and modern programming environments. We needed a more semantic integration of the artifacts and an 'active' object-oriented CASE environment which monitors all the decisions made and warns us, if necessary.

8. Conclusions

The contributions of this paper is twofold: the architecture concept, and the experience that we gained in building three object-oriented frameworks. We extensively tested these frameworks from the perspective of robustness and adaptability. For example, we tested the transaction framework with dynamically changing serialization and recovery semantics. In addition, to test our implementations to somewhat 'unforeseen' changes, we asked students to apply and extend the frameworks by using, if possible, other techniques than that were implemented. For example, in [Visser 94], students extended the knowledge domain Scheduler by a hierarchical locking scheme which was not considered initially in the transaction framework [Tekinerdogan 94]. Our conclusion is that the architecture concept is the right choice. Most of the experienced obstacles were related to the problems presented in the previous section. We believe that these problems, in general, are not inherent to the object-oriented concepts but rather they relate to the way how concepts are defined and implemented in the current methods and programming languages. We are, therefore, optimistic in that most of the presented problems in this paper can be solved, at least partially.

References

[Abreu 94] F. Brito e Abreu, *Object-Oriented Software Design Metrics*, OOPSLA' 94 Metrics Workshop, 1994.

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

[Aksit 92a] M. Aksit, L. Bergmans & S. Vural, *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, ECOOP '92 Conference Proceedings, LNCS 615, Springer-Verlag, 1992, pp. 372-395.

[Aksit 92b] M. Aksit & L. Bergmans, *Obstacles in Object-Oriented Software Development*, Proceedings OOPSLA '92 Conference Proceedings, ACM SIGPLAN Notices, Vol. 27, No. 10, October 1992, pp. 341-358.

[Aksit 93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, Abstracting Object Interactions Using Composition-Filters, in ECOOP'93 Workshop Object-Based Distributed Programming, (eds) R. Guerraoui et al, LNCS 791, Springer-Verlag, July 1993, pp. 152-184.

[Aksit 94] M. Aksit, J. Bosch, W. van der Sterren, L. Bergmans, Real-Time Specification Inheritance Anomalies and real-Time Filters, ECOOP'94 Conference Proceedings, LNCS 821, Springer-Verlag, 1994, pp. 386-407.

[Aksit 96] M. Aksit & F. Marcelloni, *Minimizing Quantization Error and Contextual Bias Problems of Object-Oriented Methods by Applying Fuzzy-Logic Techniques*, Draft paper, University of Twente, 1996.

[Bergmans 95] L. Bergmans & M. Aksit, *Composing Real-Time and Synchronization Constraints*, *accepted for publication for the Journal of Parallel and Distributed Computing*, Memoranda Informatica 95-41, University Of Twente, November 1995.

[Bernstein 87] P.A. Bernstein, V. Hadzilacos, & N. Goodman. *Concurrency control and recovery in Database Systems.* Addison-Wesley, 1987.

[Booch 91] G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc, 1991.

[Carey 86] M. Carey, & W. Muhanna. *The performance of multiversion concurrency control algorithms*, ACM Transactions on Computer Systems, Vol. 4. No. 4, November 1986, pp. 338-378.

[Chidamber 94] S. R. Chidamber & C. F. Kemerer, *A Metrics Suite for Object-Oriented Design*, IEEE Transactions on Software Engineering, Vol. 20, N. 6, 1994, pp. 476-492.

[Coad 91] P. Coad & E. Yourdon, *Object-Oriented Analysis*, 2nd edition, Yourdon Press Computing Series, Prentice-Hall, 1991.

[Dubois 80] D. Dubois & H. Prade, *Fuzzy Sets and Systems*, Academic Press, 1980.

[Ferber 89] J. Ferber, *Computational Reflection in Class-Based Object-Oriented Languages*, OOPSLA '89 Conference Proceedings, ACM SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 317-326.

[Gamma 95] E. Gamma, R. Helm, R. Johnson & J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Garlan 95] D. Garlan & D. E. Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transasctions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 269-274.

[Guerraoui 94] R. Guerraoui, *Atomic Object Composition*, ECOOP '94 Conference Proceedings, LNCS 821, Springer-Verlag, 1994, pp. 118-138.

[Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson & G. Overgaard, *Object-Oriented Software Engineering -- A Use Case Driven Approach*, Addison-Wesley/ACM Press, 1992.

[Johnson 88] R. Johnson & B. Foote, *Designing Reusable Classes*, Journal of Object-Oriented Programming, June/July 1988, pp. 23-35.

[Klir 88] G.J. Klir & T.A. Folger, *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, Canada Inc., Toronto, 1988.

[Lee 90] C.C. Lee, *Fuzzy Logic in Control Systems: Fuzzy Logic Controller, Part I-II*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 20, No. 2, March/April, 1990, pp. 404-435.

[Levesque 79] H. Levesque & J. Mylopoulos, A Procedural Semantics for Semantic Networks, in Associative Networks, (eds) Findler, 1979, pp. 93-119.

[Lieberman 86] H. Lieberman, *Using Prototypical Objects to Implement Shared Behavior*, OOPSLA '86 Conference Proceedings, ACM Sigplan Notices, Vol. 21, No. 11, November 1996, pp. 214-223.

[Mandami 77] E.H. Mandami, *Application of Fuzzy Logic to Approximate Reasoning using Linguistic Synthesis*, IEEE Transactions on Computer, Vol. C-26, No. 12, 1977, pp. 1182-1191.

[Marcelloni 95a] F. Marcelloni & M. Aksit, *The Design and Application of an Object-Oriented Fuzzy-Logic Reasoning Framework*, Draft paper, University of Twente, 1995.

[Marcelloni 95b] F. Marcelloni, *On Inferring Reasonable Conclusions for Fuzzy Reasoning with Multiple Rules*, Draft paper, University of Twente, 1995.

[Minsky 75] M. Minsky, A Framework for representing Knowledge, In the Psychology of Computer Vision, (eds) P. Winston, McGraw-Hill, 1975, pp 211-277.

[Moss 85] J.E.B. Moss, *Nested Transactions: An Approach to Reliable Computing*, MIT Press, 1985.

[Mullet 95] P. Mullet, J. Malenfant and P. Cointe, *Towards a Methology for Explicit Composition of MetaObjects*, OOPSLA'95 Conference Proceedings, ACM Sigplan Notices, Vol. 30, No. 10, October 1995, pp. 316-330.

[Nierstrasz 95] O. Nierstrasz & D. Tsichritzis (eds), *Object-Oriented Software Composition*, Prentice Hall, 1995.

[OOPSLA_Panel 95] S. Gossain, M. Lubards, E. Seidewitz, H. Gomaa, D. Batory & C. Pidgeon (panelists), OOPSLA '95 panel *Objects and Domain Engineering*, Reliable Communications, tape 20, 1995.

[OOPSLA_Workshop 95] H. Segel, S. Fraser, J. Coplien, B. Castor & J. White (Org.), OOPSLA '95 workshop on *Application of Domain Analysis Techniques to Object-Oriented Systems,* 1995.

[Prieto-Diaz 91] R. Prieto-Diaz & G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Ritter 87a] G.X. Ritter & P.D. Gader, *Image Algebra techniques for Parallel Image Processing*, Journal of Parallel and Distributed Computing, No. 4,1987, pp.7-44.

[Ritter 87b] G.X. Ritter, M.A. Shrader-Frechette & J.N. Wilson, *Image Algebra: A Rigorous and Translucent Way of Expressing All Image Processing Operations*, in Proceedings of the 1987 SPIE Tech. Symp. Southeast on Optics, Elec.Opt. and Sensors, orlando, FL, May 1987.

[Ritter 90] G.X. Ritter, J.N. Wilson & J.L. Davidson, *Image Algebra: An Overview*, Computer Vision, Graphics and Image Processing, 49, 3, 1990, pp. 297-331.

[Rumbaugh 91] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Shaw 95] M.Shaw, *Making Choices: A Comparison of Styles for Software Architecture*, IEEE Software, Vol 12, No 6, November 1995, pp. 27-41.

[Tekinerdogan 94] B. Tekinerdogan, *The Design of an Object-Oriented Framework for Atomic Transactions*, Msc thesis, University of Twente, Dept. of Computer Science, The Netherlands, 1994.

[Turksen 93] I.B. Turksen & Y. Tian, *Combination of rules or their consequences in fuzzy expert systems*, Fuzzy Sets and Systems, Vol. 58, 1993, pp. 3-40.

[Visser 94] B.S. Visser, M. J. Evers and C.W. van den Ende, *A Multi User Software Development Environment Framework in Smalltalk*, Design Project, University of Twente, November 1994.

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

[Wegner 84] P. Wegner, *Capital-Intensive Technology and Reusability*, IEEE Software, July 1984, 1984, pp. 7-45.

[Weihl 89] W.E. Weihl. *Local atomicity properties: Modular concurrency control for abstract data types*, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 2, April 1989, pp. 249-282.

[Zadeh 73] L.A. Zadeh, *Outline of a New Approach to the Analysis of Complex Systems and Decision Processes*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, No.1, January, 1973, pp. 28-44.

[Zadeh 85] L. A. Zadeh, *Syllogistic Reasoning in Fuzzy Logic and its Application to Usuality and Reasoning with Dispositions*, IEEE Transactions on Systems, Man, and Cybernetics, Vol.SMC-15, No.6, Novermber/December, 1985, pp. 754-763.

[Zimmermann 91] H.J. Zimmermann, *Fuzzy Set Theory and its Applications*, Second Edition, Kluwer Academic Publishers, Boston/Dordrecht, London, 1991

[Zwet 94] P.M.J. van der Zwet, *Work Breakdown Structure Knowledge Guided Image Processing*, Internal Memo, LKEB, University Hospital, Leiden, 1994.

Appendix: Object Models

A. Transaction Framework

In this section, for illustration purposes, we will explain the inheritance hierarchy of schedulers. More detailed information can be found in [Tekinerdogan 94].

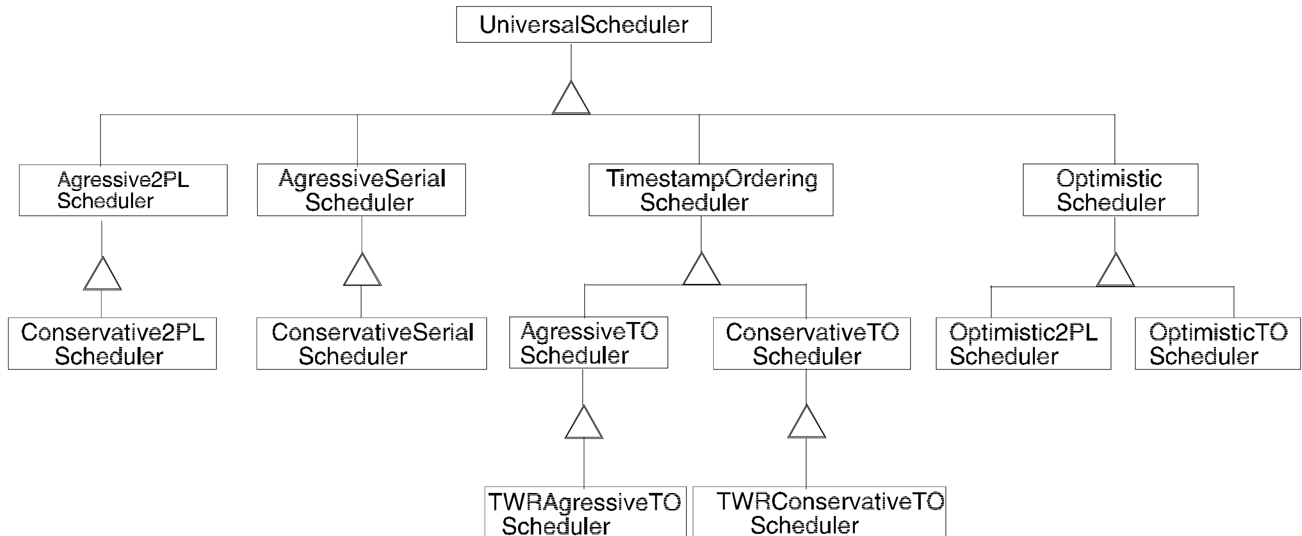

Figure 6. The inheritance hierarchy of the scheduler.

Class *UniversalScheduler* defines the common attributes and behavior for all schedulers. Class *AgressiveSerialScheduler* allows only one transaction at a time to access the object. If another transaction tries to access the same object the scheduler will immediately abort the latter transaction. Class *ConservativeSerialScheduler* is a specialization of *AgressiveSerialScheduler* but it delays a conflicting transaction until the other transaction has finished. Class *Agressive2PLScheduler* is a locking scheduler which does not delay messages in case of conflicts. *Conservative2PLScheduler* delays conflicting operations. If operations of two different transactions are mutually waiting for each other a deadlock may occur in the system. In order to resolve the occurred deadlock, the conservative locking schedulers may use deadlock avoidance and detection techniques [Bernstein 87]. The timestamp ordering schedulers may also be classified as aggressive and conservative schedulers. In both cases the Scheduler can use the Thomas-Write rule to omit a late write operation which would not have any effect at all. Optimistic schedulers either use timestamps or locks which are only controlled at commit time. In this hierarchy, classes *UniversalScheduler*, *TimestampOrdering-Scheduler* and *OptimisticScheduler* are abstract classes.

B. Image Processing Framework

The implementation of the image processing framework is shown in Figure 7. Class *Set* is the root class of the hierarchy. Class *Homogeneous* ensures that each element of *Set* is of a common type. Class *Ordered* enforces the ordering as defined in the semantics of an image. Class *Coordinate* is a set of indexing functions. Class *CoordinateSet* is a *coordinate-type homogeneous set*. Class *ValueSet* is a *homogeneous set* of *value*s. Class *ImageSpace* consists of one of the following classes: *Coordinate, CoordinateSet, Value, ValueSet* or *Image*. *Image* is a function from an *ImageSpace* into another *ImageSpace*. In order to express templates as images, we had to introduce a new class *ImageSpace*. This class provides mappings not only between coordinate sets and value sets, but also between any combination of coordinates, coordinate sets, values, value sets and images. Class *Template* implements the image algorithms. *ITOperations* are the possible algebraic operations between *Image* and *Template*. Classes *InvariantTemplate* and *VariantTemplate* classify the possible templates.

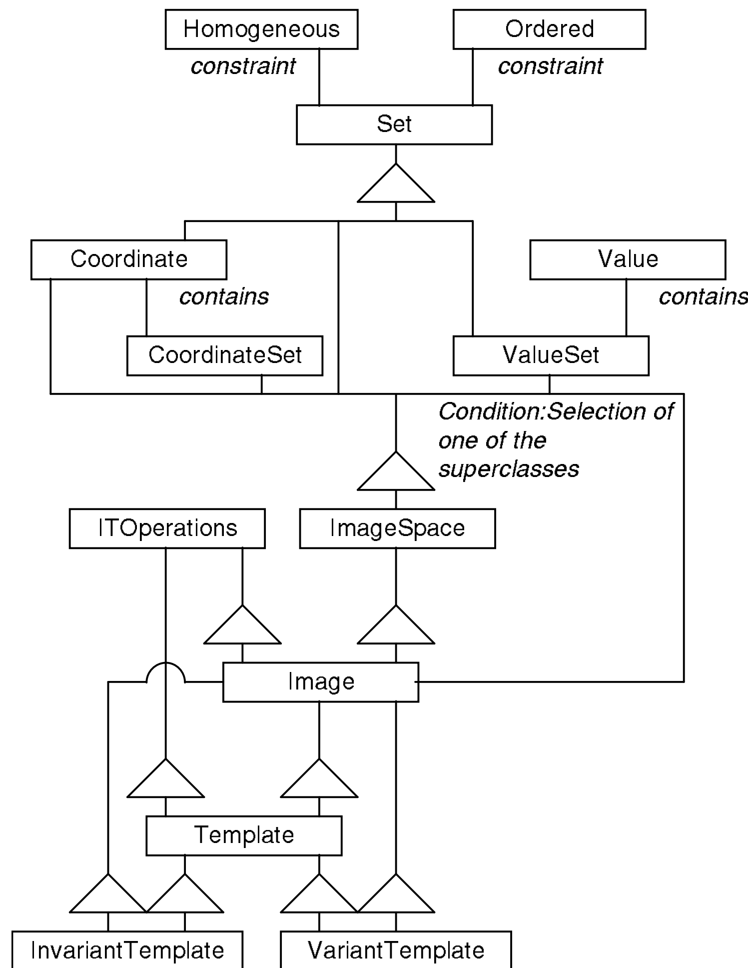

Figure 7. The object diagram of the image processing framework.

C. Fuzzy-Logic Reasoning Framework

For simplicity, we will only show the inheritance hierarchy for rules. Class *Rule* defines the common attributes and behaviors for all the possible types of rules. Class *Rule* aggregates classes *Antecedent*, *Consequent*, *ImplicationParameter* and *CompositionalParameter*. Classes *Antecedent* and *Consequent* implement the antecedent and the consequent part of a rule, respectively.

After examining the related literature, we relied that the types of implication can be grouped in three categories: *FuzzyConjunction*, *FuzzyDisjunction* and *FuzzyImplication* implications [Lee 91]. The last is in its turn divided in five families: *PropositionalCalculus*, *ExtendedPropositionalCalculus*, *Material*, *GeneralizationModusPonens* and *GeneralizationModusTollens* implications. The hierarchy in figure 8 reflects this organization. Implications differ from one another in the *triangular norm* used in the implementation. Typical triangular norms are *minimum*, *product*, *bounded product* and *drastic product*. Class *ImplicationParameter* identifies a triangular norm. Triangular norms are also used as parameters to define different possible compositional rules. During the design phase, we decided to implement the generalized modus ponens as a method of class *Rule*. Class *CompositionalParameter* identifies the triangular norm which selects the desired compositional rule. The use of parameterization reduces the hierarchy of rules.

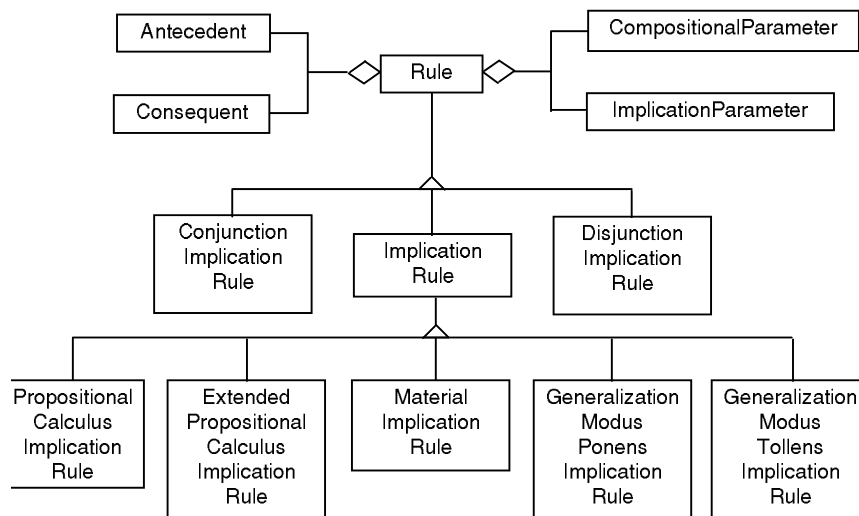

Figure 8. Rule inheritance and aggregation relations.