Deriving Object-Oriented Frameworks 
From Domain Knowledge

Mehmet Aksit\textsuperscript{1}, Bedir Tekinerdogan\textsuperscript{1},
Francesco Marcelloni\textsuperscript{2} and Lodewijk Bergmans\textsuperscript{1, 3}

\textsuperscript{1}TRESE Project, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: \{aksit | bedir | bergmans \}@cs.utwente.nl
www server: http://wwwtrese.cs.utwente.nl

\textsuperscript{2}Department of Information Engineering, University of Pisa,
Via Diotisalvi, 2-56126, Pisa, Italy.
email: france@iet.unipi.it

\textsuperscript{3}STEX bv, D.Dijkhuisstraat 248, 7558 GG, Hengelo, The Netherlands

Abstract

Although a considerable number of successful frameworks have been developed during the last decade, designing a high-quality framework is still a difficult task. Generally, it is assumed that finding the correct abstractions is very hard, and therefore a successful framework can only be developed through a number of iterative (software) development efforts. Accordingly, existing framework development practices span a considerable amount of refinement time, and it is worthwhile to shorten this effort. To this end, this paper aims at defining explicit models for the knowledge domains that are related to a framework. The absence of such models may be the main reason for the currently experienced extensive refinement effort. The applicability of the approach is illustrated by means of three pilot projects. We experienced that some aspects of domain 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 frameworks. We conclude the paper with the lessons that we learned from this experience.

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 frameworks offer well-defined infrastructures for a family of applications [Johnson 88]. Frameworks have to be tailored to the specific needs of a particular application setting, mostly through subclassing and/or compositions, preferably with minimal effort. In comparison with developing dedicated software, developing frameworks may provide long-term benefits such as enhanced productivity, reduced maintenance costs, improved consistency and better integration of software components [Taligent 96].

Although a considerable number of successful frameworks have been developed during the last several years [Apple 89], [Deutsch 89], [Huni 95], it is generally agreed that designing a high-quality framework is still a difficult task [Roberts 96], [Taligent 96]. Several methods have been proposed to support the development of frameworks. For example, in [Taligent 96], first the primary abstractions are derived from the requirement specification document and the related solutions. Second, the interactions of clients with frameworks are defined. Finally, frameworks are implemented, tested and refined.

In [Roberts 96], a pattern language is proposed for developing frameworks. The assumption made here is that primary abstractions are very hard to find, and therefore a successful framework can only be developed after a series of (software) development efforts. First, it is advised to implement three applications that could be derived from the framework to be developed. Second, as a generalization of these applications, a so-called white-box framework has to be developed. A white-box framework is structured primarily by inheritance relations. Due to the heavy use of inheritance, it requires understanding the implementation details of the used classes. Once a white-box framework is understood sufficiently, it can be converted to a black-box framework, which is based primarily on compositions. Composition-based frameworks require less knowledge of the implementation of reused classes, provide run-time adaptability and can be easily tailored by composing objects rather than programming new subclasses.

A similar approach is taken in [Huni 95], where a white-box communication framework was converted to a black-box framework. This was possible because a considerable amount of experience was gained through the application of the initial framework. The design makes extensive use of composition-based Design Patterns [Gamma 95].

It is clear that existing framework development practices span a considerable amount of refinement time, and it is worthwhile to reduce this effort. The main reason of this extensive refinement is the lack of an integrated approach to model domain knowledge related to the framework and to map the identified domain models into an object-oriented framework. For this purpose, this paper aims at finding answers to the following questions: First, would it be possible to identify and model the necessary domain knowledge for supporting framework development? The absence of such a model may be the main reason of an extensive refinement effort. Second, what might be the obstacles that one experiences in mapping domain knowledge into object-oriented frameworks? Finally, what kind of research activities would be needed to address the identified problems, if any? This paper presents our approach and findings in this experimental research. The applicability of our approach is illustrated by means of three pilot projects.

The paper is organized as follows: The following section describes the initial requirements for the pilot projects. Section 3 explains how the related domain knowledge is identified and modeled. Section 4 describes the realization of the frameworks and the experienced problems in mapping domain knowledge into object-oriented concepts. Section 5 evaluates the approach, presents the lessons learned and gives conclusions.
2. Description of the Pilot Projects

In the following we 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. 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 either completes successfully, or has no effect on data shared with other transactions.

A car dealer management system is a data-intensive system that involves several applications with varying characteristics, operates in heterogeneous environments, and may incorporate different data formats. To achieve optimal behavior, each of these aspects may require transactions with dedicated serialization and recovery techniques. 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.

Image Processing Framework
At the laboratory for Clinical and Experimental Image Processing, located at the university hospital of Leiden, an image processing system is being developed for the analysis of the human heart [Zwet 94].

Up to now, image processing algorithms have been implemented at the laboratory using procedure libraries. For example, assume that the application of three image processing algorithms algorithm1, algorithm2 and algorithm3 on the input image produces the output image:

\[
\text{outputImage} = \text{algorithm3} (\text{algorithm2} (\text{algorithm1} (\text{inputImage})))
\]

The result of the first algorithm is the input parameter of the second algorithm and the result of the second algorithm is the input parameter of the third algorithm. Here, all cascaded input-output values must be compatible. Procedures, however, are largely dependent on the representation of the input and output values [Wegner 84]. This is problematic due to the large number of different representations for images.

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

\[
\text{outputImage} = ((\text{inputImage}.\text{algorithm1}).\text{algorithm2}).\text{algorithm3};
\]

Here, inputImage receives the message algorithm1, which results in a new image that receives the message algorithm2, and so on. Provided that each image understands these messages, one may apply the algorithms to images in any order. This means, however, that each image must define all the required image processing algorithms, which may demand a large number of method definitions.

The image processing framework must be expressive enough to construct virtually any image processing algorithm that can be used for medical imaging. Effective code reuse can simplify implementation of image processing algorithms and decrease the maintenance costs.

---

1 This project is carried out together with Siemens-Nixdorf Software Center and supported by The Dutch Ministry of Economical Affairs under the SENTER program.
Fuzzy-Logic Reasoning Framework

For several years, we have been carrying out research activities in formalizing object-oriented software development processes [Aksit 96a]. One of the problems in modeling a software development process is to represent design inconsistencies and 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 [Broekhuizen 96].

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

Consider, for example, the following rule: “If an entity is relevant in 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 “the entity is not relevant”. The software engineer may, however, conclude that the entity partially fulfils the relevance criterion, and may prefer to define the relevance of an entity, for instance, as substantially relevant. A fuzzy reasoning system can accept input values such as fairly, substantially, etc. and reason, for example, about the relevance of a class. In fuzzy logic, these values are generally represented as partially overlapping sets.

The design of the fuzzy-logic reasoning framework involves 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 provide both goal-driven and data-driven reasoning. Fourth, 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

The required features of these frameworks are quite different because they relate to different application domains. The key characteristics, however, are quite similar. Each framework must support different kinds of implementations. For example, the transaction framework must provide different serialization techniques, the image processing framework must be able to support several image processing algorithms and the fuzzy-logic reasoning framework must be able to implement different implication rules. In addition, for all frameworks, adaptability and reusability are important concerns.

3. Modeling Domain Knowledge

We first model the top-level structure of frameworks using the so-called knowledge graphs [Bakker 87]. Second, we refine each node within a top-level knowledge graph into a sub-knowledge graph called knowledge domain. Finally, we identify which nodes in a knowledge domain can be included together in the top-level knowledge graph. In the following sections, these steps will be described in more detail.

3.1 Identification of the Top-Level Knowledge Graph

Figure 1 shows a knowledge graph, which represents the related background for building a simple motorized vehicle. This graph consists of four nodes: Engine, Chassis, Break and Wheels. The relations represent the direct dependencies between the nodes. For example, Engine rotates Wheels, Break stops Wheels, Chassis carries Engine, Wheels and Break.
Finding the top-level knowledge graph of a framework requires searching the related literature and finding similarities among various publications. Each node refers to a concept that is indispensable for a given framework. The minimum configuration of a framework can be found by gradually excluding concepts until essential characteristics of the framework are left. For example, concepts that can be considered as a part of another concept, are excluded from the top-level knowledge graph.

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 the underlying techniques required to construct knowledge graphs.

A number of systems have been developed for knowledge acquisition and representation. The KADS system [Wielinga 92], for example, provides three categories called *domain knowledge*, *inference knowledge* and *task knowledge* in which the expertise knowledge is analyzed and described. The KARL system [Fensel 95] was based on the principles of the KADS system, with an emphasis on formalizing expertise models and making them operational. The basic idea behind these systems, knowledge acquisition through model construction, is similar to our approach in constructing knowledge graphs. Most of the features of these systems such as complex inference mechanisms, however, were not needed in our approach. Therefore, we preferred to adopt a much simpler knowledge representation model.

The following subsections describe the identification of the top-level knowledge graphs of our three pilot projects.

### Transaction Framework

A considerable number of textbooks and articles have been written on atomic transactions [Bernstein 87] [Elmagarmid 92]. After analyzing and comparing the literature, we noticed that most publications adopt a similar structure. Figure 2 shows a top-level knowledge graph for transaction systems.

The node *Transaction* represents a *transaction block* as defined by the programmer. The node *TransactionManager* provides mechanisms for initiating, starting and terminating the transaction. It maintains the data objects that are affected by the transaction. If a transaction reaches its final state successfully, then the node *TransactionManager* sends a *commit* message to the corresponding data objects to terminate the transaction. Otherwise, an *abort* message is sent to all the data objects to undo the effects of the transaction.
The node PolicyManager determines the strategy for optimizing the transaction behavior. In most publications, PolicyManager is included in TransactionManager. We considered, however, transaction policies as a different concern, and therefore defined it as a separate node. The node DataManager controls the access to its data object and includes the nodes Scheduler and RecoveryManager. The node Scheduler orders the incoming messages to achieve serializability. Scheduler may include deadlock avoidance and/or detection mechanisms. The node RecoveryManager keeps track of changes to the data object to recover from failures.

Image Processing Framework
The image processing framework 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 image-to-image transformations [Ritter 87a, 87b, 90]. The top-level knowledge graph of the image processing framework is derived from this theory as depicted in Figure 3.

The image processing framework consists of coordinate and value sets. Images can be expressed as a composition of these two sets. The theory of image algebra introduces the concept of image templates. A template is a specific image pattern, which is used to implement image algebra operations such as rotation, zooming and masked extraction. Using image templates, an image processing algorithm can be defined as

\[ \text{anOutputImage} = \text{anInputImage}.\text{anAlgebraicOp(aTemplate)} \]

Here, \(\text{anOutputImage}\) represents the resulting image, \(\text{anInputImage}\) is the image to be processed, \(\text{anAlgebraicOp}\) is one of the basic operations defined by image algebra, and the argument \(\text{aTemplate}\) represents the algorithm to be applied on \(\text{anInputImage}\). If templates can be derived from the user requirement 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, [Dubois 80], [Lee 90], [Turksen 93], [Zimmermann 91]). After investigating the available literature, we concluded that the knowledge graph shown in Figure 4 conforms to the concepts in most of these publications.

The node PolicyManager determines the strategy for optimizing the transaction behavior. In most publications, PolicyManager is included in TransactionManager. We considered, however, transaction policies as a different concern, and therefore defined it as a separate node. The node DataManager controls the access to its data object and includes the nodes Scheduler and RecoveryManager. The node Scheduler orders the incoming messages to achieve serializability. Scheduler may include deadlock avoidance and/or detection mechanisms. The node RecoveryManager keeps track of changes to the data object to recover from failures.
We selected the so-called generalized modus ponens (G.M.P.) as the basic inference mechanism because of its common usage in the literature. In the most general form, the G.M.P. may be expressed as follows:

For a given rule \( R = \text{IF } A \text{ THEN } B \), and a fact \( A' \), the conclusion \( B' \) is equal to \( A' \circ R \), where \( \circ \) is a composition relation between the fuzzy sets corresponding to \( A' \) and \( R \).

In Figure 4, the node Fuzzy Inference Element implements the inference mechanism. This element contains Rule, Fact, G.M.P. and Conclusion. During the initialization phase, the nodes Rule and Fact communicate with the node Linguistic Variable to create a representation of themselves in terms of fuzzy sets. These fuzzy sets are provided to the node G.M.P., which carries out the inference process and generates a conclusion. The node Conclusion combines all the outputs of the related G.M.P. nodes using the connective ALSO. The result of this combination, expressed in terms of fuzzy sets, may be ‘defuzzified’ by the node Linguistic Variable. The defuzzification operation converts the fuzzy set into a crisp value or approximates it to a linguistic value. In case of a goal-driven inference, the node Linguistic Variable requests from the node Conclusion to provide a value. In case of a data-driven inference, however, the node Conclusion delivers directly a value to the node Linguistic Variable.

Specific to our framework is the node 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. An explicit formulation of the effects of the context is therefore mandatory. The node Context is an instance of the entire fuzzy reasoning framework shown in Figure 4. Context reasons about the context information and may request to the node Linguistic Variable to modify the meaning associated with the linguistic values. Notice that the node Context may also include a sub-node Context, thereby allowing specification of the effects of the context on a context, etc. If the node Context is omitted, then the interpretation of linguistic values is fixed and cannot be changed dynamically.

3.2 Refinement of Top-Level Knowledge Graphs into Knowledge Domains

The next step is the refinement of each node in the top-level knowledge graph into a sub-knowledge graph called knowledge domain. The nodes within a knowledge domain correspond to a particular specialization in the domain and the relations typically represent generalization and specialization relations. For example, the node Engine may correspond to a sub-knowledge graph including the nodes Combustion Engine, Gasoline Engine, Diesel Engine, etc.

In a particular application setting, a node in the refinement hierarchy represents a specialization of the corresponding knowledge domain. For example, while building a specific vehicle, a node which represent the knowledge “how to build engines”, will refer to a particular engine type, such as a four-cylinder combustion engine. An application, therefore, is a composition of specializations (nodes) from the related knowledge domains.

Transaction Framework

To refine the top-level knowledge graph of the transaction framework shown in Figure 2, we investigated publications related to each node. We organized the available information for each node as a graph structure.

The node TransactionManager includes transaction management, and several different commit and abort protocols. The node Scheduler relates to concurrency control and deadlock detection techniques. The node RecoveryManager includes several recovery techniques.

In this section, for illustration purposes, we show the specialization hierarchy of schedulers in Figure 5. More detailed information can be found in [Tekinerdogan 94].

Node UniversalScheduler represents the common characteristics of all schedulers. Node SerialScheduler allows only one transaction at a time to access the object. The other schedulers use various mechanisms to preserve consistent access to the object. Node LockingScheduler represents schedulers that synchronize access to the object by using locking mechanisms in case of conflicting
operations. Node *TimestampOrderingScheduler* orders operations from transactions according to the transactions’ timestamps. This Node can be further specialized by using the Thomas-Write rule (TWR) to omit a late write operation, which would not have any effect at all. Node *OptimisticScheduler* orders conflicting transactions only at commit time. Optimistic schedulers may either use timestamp ordering or locking mechanisms to preserve consistency.

![Scheduler Hierarchy Diagram](image)

*Figure 5. The refinement hierarchy of schedulers*

In case of conflicts the Schedulers may abort or delay involved transactions. 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, schedulers may use deadlock avoidance and detection techniques [Bernstein 87]. Therefore, a specialization hierarchy for deadlock handlers has been modeled as well.

**Image Processing Framework**

We show two–related– refinement hierarchies from the image processing framework:

The node *Image* defines functional dependencies between coordinate and value sets. Similarly, the node *Template* defines functional dependencies among images. Further, this node includes knowledge about image processing algorithms. *Template* is a specialization of *Image*. Further, *Template* is classified in *InvariantTemplate* and *VariantTemplate*.

The nodes *Coordinate* and *Value Sets* represent homogeneous sets, i.e., all the set elements belong to the same type. These nodes are therefore specializations of set theory, as defined by node *Set*. By defining a small number of primitive algebraic operations on homogeneous sets, different image processing algorithms can be easily defined.

![Image Algebra Hierarchy Diagram](image)

*Figure 6. The main refinement hierarchies for image algebra.*

**Fuzzy-Logic Reasoning Framework**

We briefly summarize the nodes from Figure 4, and then look at the knowledge domain for rules in particular. The node *Linguistic Variable* represents a specialization of language theory. Its knowledge
domain therefore includes the definition of a (small) language with its syntax and semantics. The node *Fuzzy Inference Element* relates to two theories: logic theory and fuzzy set theory. During a reasoning process, the nodes *Fact, Rule, G. M. P.* and *Conclusion* interact with each other. All these nodes adopt fuzzy sets as a common data structure to exchange information. The node *Rule* defines a rule. Further, it contains the definition of the implication operator and connective AND as a fuzzy relation and a fuzzy conjunction, respectively. The node *Generalized Modus Ponens* implements the *compositional rule of inference* as a composition between two relations. The node *Conclusion* implements the aggregation operation as an intersection or union between fuzzy sets. In the literature, several implementations of fuzzy implications, conjunctions, compositions, intersections and unions have been proposed.

We only show the refinement hierarchy for rules in more detail. Node *Rule* defines the common characteristics for all the possible types of rules. After examining the related literature, we concluded that the types of rules can be grouped in three categories: *Fuzzy Conjunction, Fuzzy Disjunction* and *Fuzzy Implication* implications [Lee 90]. The latter can again be refined into five families: *Propositional Calculus, Extended Propositional Calculus, Material, Generalization Modus Ponens* and *Generalization Modus Tollens* rules. The hierarchy in figure 7 reflects this organization.

![Figure 7. The refinement hierarchy of fuzzy rules.](image)

**Overview of the Related Knowledge Domains**

Table 1 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 a stable framework infrastructure.
Table 1. Summary of the related knowledge domains.

<table>
<thead>
<tr>
<th>Pilot Project</th>
<th>Node</th>
<th>Related Knowledge Domains</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Framework</td>
<td>TransactionManager</td>
<td>commit and abort protocols</td>
<td>[Elmagarmid 92]</td>
</tr>
<tr>
<td></td>
<td>PolicyManager</td>
<td>system performance control, reliability modeling techniques</td>
<td>[Agrawal 87]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and decision making</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduler</td>
<td>concurrency control and deadlock detection techniques</td>
<td>[Bernstein 87]</td>
</tr>
<tr>
<td></td>
<td>RecoveryManager</td>
<td>recovery techniques</td>
<td>[Bernstein 87]</td>
</tr>
<tr>
<td>Image Processing</td>
<td>Coordinate Set and</td>
<td>set theory, mathematical domains, algebra</td>
<td>[Ritter 87a,b]</td>
</tr>
<tr>
<td>Framework</td>
<td>Value set</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Template</td>
<td>function theory, image representation techniques, algebra,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>image processing</td>
<td></td>
</tr>
<tr>
<td>Fuzzy-Logic</td>
<td>Linguistic Variable</td>
<td>Language theory, fuzzy set theory</td>
<td>[Zadeh 73, 96]</td>
</tr>
<tr>
<td>Reasoning Framework</td>
<td>Fuzzy Inference</td>
<td>fuzzy set theory, logic theory</td>
<td>[Dubois 80]</td>
</tr>
<tr>
<td></td>
<td>Element</td>
<td></td>
<td>[Klir 88]</td>
</tr>
</tbody>
</table>

3.3 Defining Constraints and Adaptability Space

In the final step of our approach, we identify which nodes in a knowledge domain can be included together into the top-level knowledge graph. A set of semantically correct alternatives determines here the adaptability space. Each alternative defines which specializations from different domains enforce constraints on each other, when they are included within the same framework. For example, in building motorized vehicles, a specific chassis structure must be suitable to the power of the engine used. Additional user-defined constraints may be added, for example, to restrict the scope of the framework.

Transaction Framework

In the Transaction Framework, the interaction protocols between the nodes in Figure 2 determine compatibility constraints between the specializations of the corresponding knowledge domains. For example, the commit and abort protocols of TransactionManager must be understood by the corresponding DataManager. If the protocols of TransactionManager are changed, then the protocols of the DataManager must be changed accordingly. If the transaction behavior is dynamically changed, for instance by the operating system, then the nodes Scheduler and RecoveryManager must be adapted accordingly.

In addition to interaction compatibility requirements, there may be restrictions on the composability of components. For example, the nodes Scheduler and RecoveryManager are in some cases dependent on each other [Weihl 89]. Therefore, not every node of the knowledge domain Scheduler can be combined with all nodes of the knowledge domain RecoveryManager. Finally, the different serialization protocols adopted by scheduler nodes may be incompatible with each other [Guerraoui 94].

Image Processing Framework

There are two important constraints for the elements of coordinate and value sets. First, these sets must be homogenous. Therefore, a coordinate set must only contain, for instance, 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 a 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.

Further, additional constraints are imposed by the algebraic operations. An algebraic operation between two images, for instance, may only be performed if both images have exactly the same coordinate set.
Fuzzy-Logic Reasoning Framework

The nodes of the top-level fuzzy-logic reasoning graph, as defined by Figure 4, can be considered as specializations of some aspects of fuzzy-set theory. Theoretically, we can select each combination of specializations for implementing the reasoning process. For instance, in the node *Rule*, the connective AND and the implication operator may be interpreted as a fuzzy conjunction, which uses the minimum operator, and the Mandami’s implication operator [Mandami 77], respectively. The node *G.M.P.* may be implemented by the max-min compositional rule of inference defined by Zadeh [Zadeh 73]. The node *Conclusion* may implement the connective ALSO as a fuzzy union which uses the maximum. Not all the possible combinations, however, can produce logically meaningful conclusions [Turksen 93] [Marcelloni 96]. 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 2, all the three frameworks require interaction and composability constraints to guarantee correct behavior. These constraints define the adaptability space of each framework.

<table>
<thead>
<tr>
<th>Pilot Project</th>
<th>Required Adaptability</th>
<th>Inter-node constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction framework</td>
<td>scheduling and recovery concepts</td>
<td>intra-data manager (scheduler and recovery) and inter-data manager</td>
</tr>
<tr>
<td>Image processing framework</td>
<td>different coordinate and value types, a large possible number of templates in 8 categories</td>
<td>sets must be homogeneous, ordering of elements in sets, type compatibility restrictions imposed by algebraic operations, 8 categories of templates</td>
</tr>
<tr>
<td>Fuzzy-logic reasoning framework</td>
<td>several implementations of fuzzy reasoning, language used in the rules</td>
<td>rule, generalized modus ponens and conclusion are constrained each other by logical soundness, rules and facts constrained by the linguistic variable</td>
</tr>
</tbody>
</table>

Table 2. Required adaptability and inter-node constraints.

4. Mapping Knowledge Graphs to Object-Oriented Frameworks

4.1 Experienced Problems

During the mapping of knowledge graphs to object-oriented frameworks, we experienced a number of problems because not all the elements of the knowledge graphs could be directly mapped into the object-oriented concepts. As a consequence, we were for example forced to represent some elements in the implementation of operations of objects instead of adopting explicit representations. This may reduce adaptability and reusability of frameworks. The following sections explain some significant problems that we experienced during the development of the frameworks.

4.1.1 Dynamically Changing Implementations

In many situations, the implementation of an object may not be fixed but can change at object initialization or execution time because of improvement and/or evolvement requirements. Improving can be necessary, for example, to optimize time and space performance of objects. Evolving is required for dealing with open-ended behavior of real-world systems.

In all the pilot projects, dynamically changing implementations are required. For example, in the transaction framework shown in Figure 2, the nodes *Scheduler* and *RecoveryManager* have to be adapted dynamically with respect to changing application and/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.

In the image processing framework, dynamically changing implementations are required mainly for improving time 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 time efficient for certain algorithms since each image element can be directly accessed.
In the fuzzy-logic reasoning framework, a particular implementation of the nodes from the knowledge graph affects the results of the reasoning. The type of application and the input values generally determine such a choice. Therefore, only at run-time it is possible to determine the implementation, which allows inferring the desired conclusions. For most fuzzy-logic reasoning systems, instantiation of implementations during object creation would be satisfactory. For reasoning systems with learning behavior, however, the implementation may change dynamically.

The Bridge or Strategy patterns [Gamma 95] can be used to define objects with dynamically changing implementations. In these patterns, different implementations are represented as objects. Now let us assume that $C_d$ is the class that requires a dynamic implementation. Therefore, $C_d$ encapsulates its implementation object $O_i$. Here, $O_i$ implements the methods $m_1$ to $m_n$. $C_d$ declares these methods at its interface, but redirects the requests for these methods to $O_i$ by invoking the corresponding methods on $O_i$. For example, $C_d$ implements the method $m_1$ in the following way:

```java
C_d::m1(arguments)
return O_i.m1(arguments);
```

Provided that all the implementation objects implement the methods $m_1$ to $m_n$, one can change the implementations of class $C_d$ by assigning a new implementation object $O_{new}$ to $O_i$.

```java
O_i := O_{new};
```

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

There are, however, a number of problems with this approach. First, class $C_d$ must declare all the methods $m_1$… $m_n$ explicitly. If $n$ is large, this can be a tedious and error-prone task, particularly if $O_i$ inherits a lot of methods defined in its superclasses. Second, the Bridge and Strategy patterns cannot be used for evolving systems. The precise set of methods and their arguments has to be fixed when class $C_d$ is defined since $C_d$ has to declare all the dynamically changing methods explicitly. Third, although the implementation object behaves like a superclass, it cannot polymorphically refer to the encapsulating object (instance of $C_d$) through self calls. This is similar to the self-problem as defined in [Lieberman 86].

An alternative to the pattern approach is the delegation mechanism [Lieberman 86]. If an object cannot respond to a particular request of a client, then it delegates this request to one or more designated objects. One of the designated objects may execute the request on behalf of the object. Further, the designated object can refer to the object by calling on the pseudo variable self. Delegation is similar to inheritance; the designated object behaves like the superclass of the object. Delegation can express dynamic implementations if an object delegates the requests, which it cannot respond to, to its internal implementation objects. Delegation, therefore, eliminates the need of declaring the dynamically changing methods explicitly and can support the evolution of the implementation objects. Further, delegation solves the self-problem by providing the pseudo-variable self. The conventional delegation mechanism, however, cannot enable or disable the delegation process, for example, based on a condition of the delegating object. This may be necessary, for example, if the implementation of an object has to be adapted based on a state of that object. In the pilot applications, we found a conditional delegation mechanism useful in adapting the behavior of an object in a well-defined manner. The State Pattern [Gamma 95] does not provide an adequate solution for this problem because it has the similar limitations as the Bridge or Strategy pattern.

In our prototypes, we have implemented the conditional delegation pattern using the so-called Dispatch filter [Aksit 92a]. Dispatch filter effects the incoming messages to the object that it is attached to and thereby can implement a conditional delegation mechanism.
4.1.2 Difficulties in Expressing Knowledge Specializations Using Class Inheritance

In our approach, the related knowledge domains are identified and represented by using generalization and specialization relations. We experienced that the generalization-specialization hierarchies as defined in the knowledge domains cannot always be directly mapped to the object-oriented inheritance hierarchies.

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 domains.

In the transaction framework, for instance, the PolicyManager chooses a particular policy by applying several 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.

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

Implementing a dedicated inheritance mechanism as a framework feature can solve the problem of representing knowledge specializations. In [Aksit 90], for example, a grammar inheritance mechanism is presented as a structural organization of grammar rules by which a grammar inherits rules from super-grammars or may have its own rules inherited by sub-grammars.

4.1.3 Architectural Constraints

As discussed in section 3.3, a number of constraints must be enforced upon the top-level knowledge graph. For example, nodes from different knowledge domains may not be composed arbitrarily. We consider the enforcement of such constraints as fully distinct and independent from the application behavior.

In the Transaction Framework, for instance, many different specializations are available for both the nodes Scheduler and RecoveryManager. One of the main reasons for separating the Scheduler and RecoveryManager is that these are largely orthogonal. This allows choosing independent specializations. However, in a number of cases, these domains are not orthogonal: adopting a particular type of Scheduler excludes certain types of RecoveryManager. This implies that whenever the composition is changed, the consistency of the new composition must be checked. Although the verification may involve interactions with multiple objects, its specification must be modular so that it can be adapted and reused separately from the application classes.

The enforcement of constraints on composition is 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. However, when several components and complex rules determine the constraints on composition, a more powerful type checking mechanism than subclassing and/or signatures is needed.

---

2 In [Aksit 92b], this problem was termed as “arbitrary inheritance”.

3 Note that it is usually possible to implement an object-oriented application that provides correspondence to a domain knowledge hierarchy. However, this may require the creation of additional structures and interactions because a one-to-one mapping is impossible. Usually those additional structures have a negative impact on the adaptability and extensibility.
In general case, the main difficulty is that constraint specifications are required to be modular, but at the same time the enforcement of constraints may be needed at many different locations and circumstances.

To solve these problems, in the pilot projects we have adopted meta-level objects which monitor and control the compositional structure of the architecture [Aksit 93].

4.1.4 Other Difficulties

In this section, we briefly mention two other relevant issues that we had to deal with in realizing the frameworks. 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 should be accessed in two distinct ways with respect to the type of client. The application-specific functionality should be invoked by other application objects (user view), whereas the data management functionality, such as locking or recovery methods, should be used by the transaction framework (system view). The enforcement of such distinct views, which is important for preserving 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 referred to as the shared behavior affected by shared state problem. This problem is encountered whenever a particular state shared by multiple objects affects the behavior shared by these objects. Sharing of behavior is usually achieved by a code reuse mechanism such as inheritance. Class inheritance cannot, however, adequately deal with a shared state that is encapsulated under the shared behavior. This is because each instance of a class in a class hierarchy has its own encapsulated state. Using an external server object for retrieving the shared state weakens encapsulation. In addition, the polymorphic variable self refers to the server object but not to the object that provides the shared behavior (the self problem [Lieberman 86]).

In the transaction framework, for example, all TransactionManager objects share the behavior of the PolicyManager. 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 described in more detail in [Aksit 92b].

4.1.5 Overview

Table 3 provides an overview of the pilot projects, showing where certain difficulties were encountered, with a brief description of the area.

<table>
<thead>
<tr>
<th>Pilot project</th>
<th>Dynamic implementations</th>
<th>Inheritance vs. knowledge specializations</th>
<th>Constraints</th>
<th>Multiple views</th>
<th>Sharing behavior &amp; state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction framework</td>
<td>scheduling, recovery</td>
<td>policy manager</td>
<td>data manager</td>
<td>user-system views</td>
<td>system parameters</td>
</tr>
<tr>
<td>Image processing framework</td>
<td>alternative implementations</td>
<td>no</td>
<td>value &amp; coordinate sets</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Fuzzy-logic reasoning</td>
<td>fuzzy-logic implementation</td>
<td>linguistic variables</td>
<td>Operator types</td>
<td>linguistic variables</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 3. Pilot projects versus problems.

4.2 Implementation of the Frameworks

Table 4 summarizes the most important characteristics of the implementation of the three frameworks. Here, the column Language indicates the programming language used in the implementation. The column Inheritance & # of Classes gives the number of classes defined within a specific inheritance hierarchy. The column Time Spent shows both the design effort and the implementation effort. The design effort indicates the total time spent in defining the knowledge graphs and designing the framework. The implementation effort shows the time spent for coding and
testing the framework. Finally, the column Reference shows where more details about the design and implementation of the frameworks can be found.

<table>
<thead>
<tr>
<th>Pilot Project</th>
<th>Language</th>
<th>Inheritance &amp; # of Classes</th>
<th>Time Spent</th>
<th>Reference</th>
</tr>
</thead>
</table>
| Transaction          | Smalltalk         | scheduler hierarchy: 8  
dead-lock hierarchy: 7  
recovery hierarchy: 9  
other: 20                                      | design = 6 months  
impl. = 1 months       | [Tekinerdogan 94]    |
| framework            |                   |                                                                                          |                         |                   |
| Image processing     | C++               | single inheritance hierarchy: 20                                                         | design = 6 months  
impl. = 2 months       | [Vuijst 94]          |
| framework            |                   |                                                                                          |                         |                   |
| Fuzzy-logic          | Smalltalk         | rule hierarchy: 8  
linguistic variable hierarchy: 4  
membership functions hier.: 8  
linguistic value hierarchy: 10  
other classes: 29 | design = 6 months  
impl. = 1 months       | [Marcelloni 97]      |
| reasoning framework  |                   |                                                                                          |                         |                   |

Table 4. Implementation aspects of the frameworks.

Transaction Framework
The transaction framework has been implemented using the Smalltalk language [Goldberg 83]. To change the implementations of Scheduler and RecoveryManager, we implemented a delegation mechanism on top of the Smalltalk language. Each delegated message is reified and represented as a first-class object. This 'message object’ can be treated and manipulated like other objects. In the literature, this concept is known as message reflection [Ferber 89]. By changing the attributes of a message object (in particular: the receiver of the message) and re-activating it again so that a real message invocation is created from the object, a delegation mechanism can be realized.

In the implementation of the Transaction Framework, constraints on object interactions and compositions are defined in separate constraint classes. To enforce a constraint, the messages that may violate the constraints are reified and redirected to the constraint objects. 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 node within a knowledge domain into a class. The implementation consists of 44 classes. Each knowledge domain is represented by inheritance hierarchies. The framework consists of 3 major inheritance hierarchies.

In the current prototype, class TransactionManager implements a single commit/abort protocol. Class PolicyManager adopts a simple policy management strategy. Our future work includes the implementation of 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 has been implemented using the C++ language [Stroustrup 91]. Each node of a knowledge domain is mapped into a C++ class. Similar to the transaction framework, interaction and composability constraints are enforced by defining meta-level classes, and reifying and redirecting the messages that may violate the constraints to these classes.

Currently, classes Coordinate Set, Value Set, Image and Template 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 in the Smalltalk language.
In the framework, class *LinguisticVariable* has two major methods for the fuzzification and defuzzification process. At the moment, we have implemented only the most common defuzzification strategies by using a Strategy Design Pattern. Class *LinguisticVariable* is the root of the inheritance hierarchy in which each subclass implements a different linguistic variable.

Rules are organized in an inheritance hierarchy that was shown in figure 7. Class *Fact* is composed of one or more instances of class *Proposition*. The classes *Proposition* and *Rule* inherit from class *FuzzySet* which encapsulates class *MembershipFunction*. We have defined eight types of membership functions. The node *G.M.P.* 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. So far, we have only considered the generalized modus ponens as a fuzzy reasoning mechanism. We intend to investigate other possibilities such as the syllogisms proposed by Zadeh [Zadeh 85]. Further, we will implement more defuzzification strategies and membership functions. Alternative implementations of the generalized modus ponens which can be used with particular implication operators and fuzzy sets are being analyzed (see e.g. [Broek 97], [Lazzerini 97]). Such implementations allow for considerable performance improvements.

**Comparison of the Implementations**

All three implementations have been derived directly from the respective knowledge domains. In addition to adopting 'standard' object-oriented models and design patterns, delegation and message reflection techniques were implemented to increase software adaptability and reusability. In all the pilot projects, the designers were not experienced in the corresponding domains. Therefore, they spent a considerable amount of their time in understanding the related domains knowledge.

5. **Evaluation of the Approach and Conclusions**

The main claim of this paper is that the framework refinement through iteration effort may be reduced considerably by modeling the related domain knowledge explicitly. In other words, black-box frameworks can be derived directly by composing a number of specializations from the related knowledge domains. To verify this claim, we proposed a framework development approach based on modeling domain knowledge, and carried out three pilot projects.

We extensively tested these frameworks from the perspective of robustness and adaptability. For example, we tested the transaction framework for dynamically changing serialization and recovery semantics. In addition, to test our implementations on ‘unforeseen’ changes, we asked students to apply and extend the frameworks by using, if possible, techniques different from those already implemented. For example, in [Visser 94], students successfully extended the knowledge domain Scheduler with a hierarchical locking scheme which was not considered initially in the transaction framework [Tekinerdogan 94].

Our conclusion is that modeling domain knowledge explicitly may reduce the number of iterations and the amount of refinement time required for achieving stable, robust frameworks. However, we experienced some problems when we tried to map domain knowledge to the object-oriented model. Our findings are summarized by the following items:

- **Specific inheritance semantics are necessary for certain knowledge domains**: The method and attribute inheritance mechanisms as defined by most object-oriented models are not always suitable 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. An example of such an approach is the grammar inheritance mechanism [Aksit 90].

- **Conditional delegation is needed**: As discussed in section 4.1.1, we found the delegation mechanism quite necessary in defining adaptable software systems. For example, delegation can help in improving several design patterns such as *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 4.1.4). We do not consider, however, delegation as a replacement of inheritance or class concepts. Both delegation, inheritance and class concepts can coexist together. A major problem of the conventional delegation mechanism is that the delegation relations of an object can not be controlled, for example based on a state of the object. We, therefore, extended the conventional object model using Dispatch filters [Aksit 92a].

- **Enforcing constraints is essential, but not fully supported yet:** To instantiate and manage a dynamically evolving application while preserving its robustness, high-level mechanisms to enforce the semantic constraints of that application are required. 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. A possible approach to solving this problem is to introduce meta-level objects which monitor and control the compositional structure of the application. To this aim, we used meta-objects called Abstract Communication Types [Aksit 93]. However, in our current implementations, meta objects to enforce constraints only offer run-time verification.

- **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 [Aksit 96b], [Bergmans 96], [Kiczales 97], [Mullet 95], [Nierstrasz 95]. Since frameworks can be considered as a composition of specializations from 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 frameworks [Bergmans 97].

In our own work, to separate concerns from each other, we developed the composition-filters model [Aksit 92a, 93, 94]. Composition filters extend the conventional object model in a modular and composable way. A modular extension means that the basic characteristics of the underlying language model remain the same. A composable extension means that various filters can be attached to the same object independently, because filters are semantically orthogonal to each other. For example, dynamic delegation and constraint enforcement can be added to an object by simply attaching a Dispatch and a Meta filters together.

- **Software artifacts must be recorded, related and integrated:** During the software development process, from domain analysis to coding, lots of information was 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.

To overcome these problems, during the last several years we have been carrying out research activities in modeling software artifacts and design environments [Aksit 97]. Basic concepts underlying this research are to make each software artifact self-contained, with its own ‘intelligence’ in the form of rules, and active in the sense that each artifact will initiate activities to keep the software system under development correct and up-to-date. Further we apply fuzzy logic techniques to define rules, since this allows expressing design heuristics in a more accurate way than traditional two-valued logic [Aksit 96a].

Most of the above problems can be attributed to the lack of expressiveness of the object-oriented model. By adopting the solution approaches as described for each of the above items, we were able to successfully realize the pilot frameworks. Concluding, we are convinced that the approach described in this paper allows reducing the traditional number of refinement steps by developing frameworks directly from domain knowledge.
References


