RAACR: A RECONFIGURABLE ARCHITECTURE FOR ADAPTING TO CHANGES IN THE REQUIREMENTS

A Thesis Presented
by

Yönet Arif Eracar

to
the Graduate School of Engineering

in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

in the field of
Computer Systems Engineering, Engineering Software Design

Northeastern University
Boston, Massachusetts

September 1996

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: RAACR: A Reconfigurable Architecture for Adapting to Changes in the Requirements.

Author: Yönet Arif Eracar.

Program: Computer Systems Engineering (Engineering Software Design)

Approved for Thesis Requirements of the Master of Science Degree:

Thesis Advisor: Mieczyslaw M. Kokar	Date
Thesis Reader: Sagar V. Kamarthi	Date
Chairman of Department: John W. Cipolla	Date

Graduate School Notified of Acceptance:

Director of the Graduate School: Yaman Yener	Date

Copy Deposited in Library:

Reference Librarian | Date

Abstract

The subject of this thesis is a domain-specific (object/target recognition) software system (RAACR) that can adapt to changes in software requirements through the incorporation of feedback. RAACR is a hierarchy of domains (blackboards) each including multiple knowledge sources (KSs) with generic feedback estimation and adaptation mechanisms. Domain schedulers incorporate an adaptable KS scheduling mechanism which provides on-line restructuring of the processing. A generic communication mechanism is implemented on the CORBA compliant SPRING operating system. The adaptability of the system is evaluated quantitatively using a requirements volatility measure and the probability of correct recognition. Finally, the performance of RAACR is compared to the performance of a non-adaptable version of the system in an object recognition scenario.

Acknowledgements

I had faced a few challenges before I came to the United States. But none of them was as stressful and painful as my studies for getting this Masters of Science degree. Not only the research and the courses but also living in a new country, following a new life style and many unexpected, undesired events caused a long delay in completion of this thesis.

Here, I want to thank two special persons, Prof. Mitch Kokar and Prof. Yaman Yener. They provided the environment, supported my education and directed my research. They directly or indirectly changed my life.

Also, I want to thank Prof. Sagar Kamarthi. He accepted to review my thesis without hesitation as a last minute obligation.

I want to thank my dearest wife Elif for supporting me and standing by me. She never gave up believing in me, even at the moments I gave up.

Thank you, thank you my friends at Northeastern. Dinçer, Kıvanç, Aybek, Omca, Sibel, İlker, Emre, Ozan, Erozan, Fikri, Ayşe, Zbigniew, Stephen, Aşkıner. You listened to my complaints when I was angry. You gave me advice when I was clueless. And you shared my fears when everything seemed to be dark.

Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

In object/target recognition applications, it is difficult to predict all possible scenarios that the sensors would have to deal with in the future. In addition to this, the signal processing hardware itself, due to the complexity of processing, would introduce all kind of randomness in the sensory data. Consequently, object/target recognition software is built without complete specification of inputs, functionality and constraints. The result is that such a software fails to achieve its goal, i.e., recognize objects/targets when unexpected changes in the inputs occur.

Control engineers are faced with a similar problem: they have to design controllers that can deal with unexpected disturbances. The methods they employ to solve this kind of problem are *feedback*, *adaptation* and *reconfiguration of the controller*.

The main goal in this thesis is to develop a domain-specific (object/target recognition) software system that incorporates some of the ideas of adaptable and reconfigurable control for dealing with the issue of incomplete specification of requirements, it is a software system that is adaptable to changes in the specifications. The incompleteness of specification for the domain we deal with in this thesis may include: inputs (binary vs. gray scale images, types of objects in the images), functionality (which of the image processing functions to use), and constraints (alignment of edges with image borders vs. unconstrained orientation of edges).

Although we are dealing with a specific domain, our intention is to investigate generic methods for building software systems that can adapt to changes in the requirements through the incorporation of feedback information in their processing. Consequently, the mechanisms we investigate in this thesis are generic mechanisms (mainly statistical algorithms) rather than specific to the domain of object recognition. We implement them in a system that we call a *Reconfigurable Architecture for Adapting to Changes in the Requirements (RAACR)*.

Our object recognition application, similarly as many other applications, is composed of different tasks, each accomplished in a different software componenent. These components are organized into a number of abstraction levels. Each level includes a group of components with similar tasks and data types. The resulting architecture is a multi-level blackboard.

The main reasons for this kind of hierarchical organization are to reduce the complexity of processing. For example, object recognition on raw data, although possible, can not be performed efficiently. Instead, feature extraction takes place first, and then, features are used for object identification. Feature extraction is an example of data abstraction. Often multiple consecutive abstractions need to be performed before objects can be recognized. This results in a hierarchy of processing. The hierarchy of multiple levels allows pipelining which is suitable for implementation on a multiprocessor architecture and it can result in a fast computation.

A system with multiple abstraction levels may consist of many processes running sequential or parallel, communicating, cooperating or competing. Therefore, there is a need for a *software bus* to provide communication within the levels and among the levels. In order to provide for portability and interoperability of the RAACR, we selected a solution that is based on the CORBA (Common Object Request Broker Architecture) standard defined by the Object Management Group (OMG) [19].

The components of our application (tasks) will be called (in the RAACR) *knowledge sources (KSs)*; each of them implements a specific function. These components are organized into a number of abstraction levels, called *domains*. Each domain includes a group of KSs with similar data types and similar functionality. Additionally, each domain includes a *domain scheduler (DS)*, which implements the scheduling of the KSs. Each domain includes also a number of data transfer and initialization modules (DTS), which initialize the domain data and processing parameters of KSs. Each of the KSs may obtain feedback, either from the user or from other domains. Each KS has an adaptation mechanism that adjusts processing parameters in response to the feedback. To provide means for dealing with structural changes in the inputs to the system, when adaptation alone may be insufficient to maintain the performance of the system, domain schedulers incorporate an adaptable KS scheduling mechanism which provides on-line restructuring of the processing. This mechanism changes its KS scheduling policy in response to the feedback obtained from the knowledge sources and other domains of the RAACR. This results in a real-time restructuring of the processing. In order to provide a generic (domain independent) communication mechanism between the levels, and to provide for portability and interoperability of the RAACR, the communication among the RAACR domains uses the Object Request Broker (ORB) of the CORBA standard [19]. The system is implemented on the CORBA compliant SPRING operating system [11, 12].

To evaluate the ability of the system to adapt to changing requirements from the software engineering point of view, *volatility of requirements* is measured using the Total Requirements Volatility measure [4]. The performance of the system for particular volatility levels is measured in terms of the probability of correct recognition. The dependence of the system performance on the volatility of the requirements is obtained and represented in the form of a bar chart diagram.

The research performed in this thesis includes a review of the existing intelligent control architectures and software architectures. An analysis of these architectures' deficiencies in real-time uncertain environments is given in Chapter 2. This review places special emphasis on the domain of sensor/data fusion and object recognition.

In Chapter 3, a description of the RAACR is given. First, a problem statement is provided to define the goals of the RAACR. Second, the architectural constraints related to the goals of the RAACR are explained. Third, the software architecture is discussed and information about objects, methods, communication mechanisms, and adaptation mechanisms is presented. An overview of the implementation of the RAACR is also provided. The system was implemented on CORBA compliant SPRING operating system [11], [12].

In Chapter 4, an experimental scenario is introduced. The initial goal (initial requirement) of the system is to recognize square shaped objects in binary images. A software system is designed and implemented according to the descriptions of the RAACR given in Chapter 3. Domains, objects, interface definitions and functionality of object methods are defined.

In Chapter 5 three increments in the requirements for the experimental scenario (object recognition) are defined. The requirements increments are then evaluated to obtain the measure of volatility of requirements. This volatility measure is used to evaluate the adaptability and the performance of the RAACR as the requirements change. A Requirements Traceability Matrix and a Requirements Graph are also provided to trace the effects of the changes. The effects of changes are described in Chapter 6.

In Chapter 6, the test steps are described. Different sets of input images are used in the tests to simulate possible changes in the environment and to evaluate the adaptation of the RAACR in response to the unexpected requirements changes. These changes included: rectangles and triangles which may be difficult to differentiate from squares, rotated squares, switching from binary to gray scale images and adding noise to input images. The results of the experiments are presented in graphical form. The results show that, due to the adaptation/restructuring mechanism of the RAACR, the system is able to adapt to changes in the requirements. The performance of the system slightly degrades, but it does not fail, which would be the case if the system did not have the adaptation capability.

Finally, in Chapter 7 we analyze the results and draw conclusions. We identify elements of the RAACR that are generic and propose them as a possible solution for building adaptable software systems consisting of multiple software components organized on a number of abstraction levels. We also identify problems that we encountered during this investigation and propose directions for future research.

Chapter 2

LITERATURE REVIEW

2.1 JDL Architecture

As stated in [10] and [17], fusion problems can be classified into three groups: (1) architectures of sensor data fusion systems, (2) world-model representations of the fusion results, and (3) techniques applied to fusing data.

The issue of the architecture for a sensor data fusion system has been recognized by both the research and the DOD sectors. Under the guidance of the Department of Defense, the Joint Directors of Laboratories (JDL) has created the Data Fusion Subpanel (DFS), which has developed the architecture known as the JDL DFS architecture. The JDL architecture is a layered architecture. It includes four levels. Level 1 includes sensor type and quality assessment, measurement alignment and correlation, state estimation, and object identification. Level 2 deals with the battlefield-related information processing. Level 3 deals with threat assessment, and Level 4 is supposed to deal with sensor management. While this architecture was very useful for the purpose of unification of various views of sensor data processing and partially for the reusability of software, it did not answer many of the important architectural questions. Essentially, this architecture is a reference model that can be used as a language of characterizing specific architectural solutions, rather than for defining such solutions. Even if the constraints specified in the JDL architecture are followed during the design, many architectural questions remain unanswered.

2.2 Hierarchical Architecture

The traditional hierarchical architecture decomposes the system into functional models. Models are organized in abstraction layers as it is shown in Figure 2.1. The system is under centralized control with top-down design. The higher-level module gets information from the lower-level module, integrates information and sends the results to the upper level. The planning operates on a more abstract level and makes decisions about how to control the whole system.

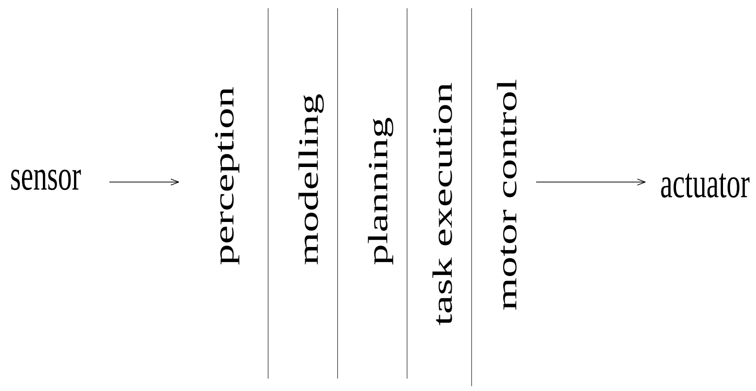

Figure 2.1: **Hierarchical Architecture**

One of the distinguishing aspects of this architecture is that the processing path is prespecified at the design stage and is strictly followed in the processing. This is advantageous, especially when all possible inputs can be foreseen during the design. However, this strictly followed hierarchical approach is not appropriate in the case of systems with high uncertainty. In systems that need to deal with a high level of both parametric and structural uncertainty, a more flexible solution is needed. One of the aspects of such a solution is its flexibility in terms of the order of application of particular processing blocks and in terms of conditions under which particular modules are invoked.

2.3 Subsumption Architecture

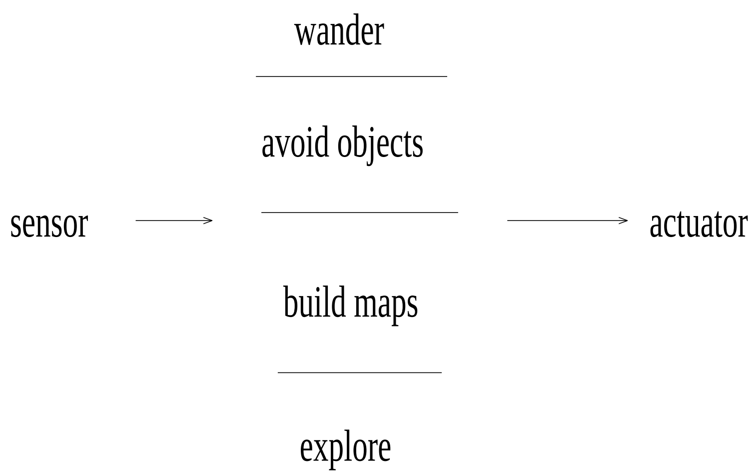

Figure 2.2: **Behavior-based subsumption architecture for a robot**

The subsumption architecture [2], also called behavior-based architecture, addresses these two issues in a manner similar to the way a CPU handles its multiple interrupts. The subsumption architecture is organized as a layered set of task-achieving modules as it is shown in Figure 2.2. Each module performs one specific behavior, such as "avoid objects", "wander". The system is under decentralized control with bottom-up design. The overall control is divided into a number of smaller, simpler, concurrent, parallel processes. So each of the modules needs to implement its own perception, modeling and planning tasks. In Brooks' subsumption architecture, agents can inhibit or suppress each other, where each sensor competes for the common goal.

As we can see, in the subsumption architecture the order of invocation and the conditions under which this invocation takes place are not prescribed. Instead, a policy of the invocation is implemented. It can be either hard-wired into the system, or it can be programmed by the user.

However, fusion is a more sophisticated method, for which strict competition by the means of priority-based interruptions and inhibition, is not sufficient. To implement fusion, there is a need to have multiple agents cooperating towards the achievement of a common system's goal. Consequently, the flexibility of the subsumption architecture should be extended.

2.4 Blackboard Architecture (BB)

The Blackboard(BB) Model is an organization of information and knowledge needed to solve a problem. It describes a dynamic control scheme and uses its knowledge for incremental, opportunistic problem solving. It may consist of multiple layers organized in a hierarchy. The important aspect of this architecture is that each layer in the hierarchy is related to the layers above and below it through the relation of abstraction.

For each function, there is one or more separate processes. In AI terms, such processes are called Knowledge Sources (KS). Each KS takes its data from a common bus and sends its outputs via the same bus. The problem-solving state data are kept in a global data store(mostly a shared memory for the modules), the *Blackboard* (BB). Communication and interaction among the KSs take place solely through the BB. KSs produce changes to the BB, which lead incrementally a solution to a problem.

KSs do not have their own decision mechanisms that allow them to make both the selection and the invocation decisions. The *controller* provides the decision mechanism to select which KS and when to invoke and which data to process. The controller monitors the BB and reacts to the changes in the BB. In most cases, multiple KSs can deal with the same input and can produce the same type of output. Therefore, KSs can compete. The controller uses its knowledge and the past statistics about the KSs to choose the KS that will give the best output for the system's goal.

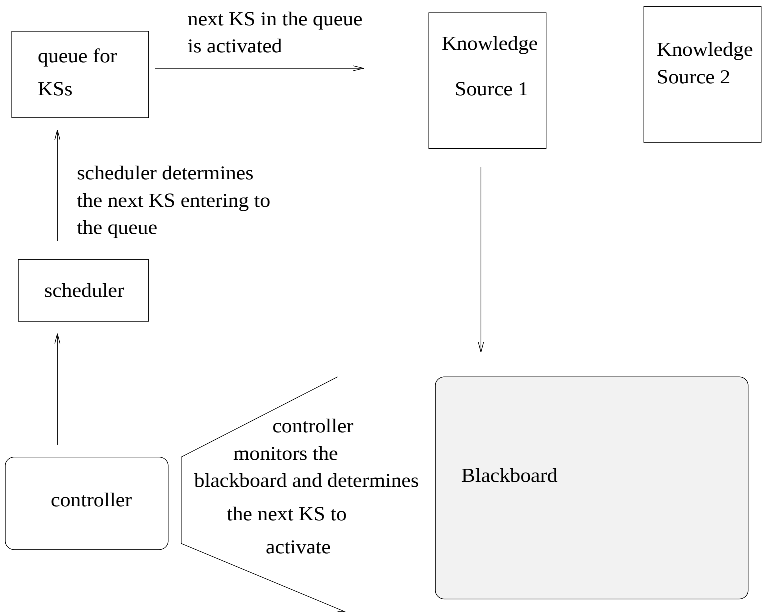

Figure 2.3: **Control flow for a simple level of blackboard**

The selection and application of knowledge sources are dynamic and opportunistic rather than fixed and programmed.

Hearsay-II, the Speech-Understanding System, was one of the first architectures implementing Blackboard Model [8]. In a single level of Hearsay-II, the flow of the control occurs as in Figure 2.3:

- The *blackboard monitor* (usually, the controller) monitors the data blackboard.
- The blackboard monitor generates rules in response to new events.
- The *goal processor* (can be a part of the controller) runs a precondition procedure to examine the applicable KSs.
- The goal processor chooses the KS with sufficient information to satisfy the goal. The goal triggers a KS instantiation (KSI).
- The scheduler assigns the KSI a priority rating and places it on the scheduling queue. Rating is done by the predicted output of KSIs.
- If a KSI is not sufficient enough, the goal processor creates goal-directed goals (subgoals) to generate the missing data.

Blackboard systems have many advantageous characteristics:

Blackboard data is organized in an hierarchical organization. The data objects on the BB are organized into levels of analysis and abstraction. Thus, different vocabularies associated with different concepts can be made explicit and accessible at the appropriate levels of abstraction. Since the layers of the system are organized according to the abstraction relation, this hierarchy imposes some constraints on the KSs: they can process only the information that is at the same abstraction level. Not only does this reduce the amount of complexity of the processing, but it also guarantees that the information that the KSs deal with is semantically in the same category. In multi-level blackboards, the communication between two levels are provided by message passing. The KSs' access to other blackboards are restricted. Thus, it causes time delays.

Blackboards realize event-driven processing. An event is any change in the information on the BB. Thus, any change in the BB, whether due to new data or to knowledge source action, is immediately recorded and made available to the controller.

Blackboards partition the knowledge. To make complex problems manageable, they need to be decomposed into loosely coupled subproblems. One useful decomposition tool is partitioning the solution space into a hierarchy, another is modular representation of knowledge sources.

Blackboards may use unreliable data and knowledge. Uncertainty in the knowledge has been resolved by assigning credibility weights to the knowledge and the information it generates. In addition to the use of diverse sources of information, different experts(competing KSs) can be used.

There is cooperation among the modules. KSs with independent or semi-independent pieces of knowledge cooperate in forming a solution.

However, there are open issues in the blackboards which should be solved. For example, control flow in one blackboard level is sequential, concurrency is not supported. In multi-level blackboards, KSs in different levels may work in parallel. But, KSs in the same level works sequentially. To realize concurrency within the blackboard levels, Inter Process Communication protocols should be defined clearly and critical section problems should be solved. A central controller or a scheduler at each blackboard level schedules the invocation order of the KSs. The controller decides when and which KS to invoke. This forces the controller to keep functional information about each KS. Any modification in a KS causes a modification in the controller, which makes transparent insertion of new KSs to the blackboard impossible. Finally, real-time applications are not supported thoroughly. Interrupt handling is not provided.

2.5 Hardware Solutions

The solutions discussed so far concern software design. Sensor data fusion is a computationally intensive process. Thus, in order to implement fusion in real time, appropriate hardware configuration is necessary; conventional hardware like PCs or workstations are not sufficient. Numerous efforts have been undertaken to develop hardware for supporting such computationally intensive processing tasks as signal processing, image processing, sensor fusion.

Practical implementations of sensor fusion utilize compact multi-computers which use dozens of processors to process large amounts of data in real-time. An example of these multi-computers is Karmarkar's modified systolic stereo vision system, using two cameras [15]. The system enters parallel input to each computing node. Image frame data are distributed directly from the sensors to the memories of the processors. The organization of the memory is based on virtual processing regions. The main idea behind this parallel, multiprocessor architecture is obtaining adequate computing speed to achieve real-time processing. However, these studies and solutions are problem-specific, aiming to get fast computational speed, regardless of complexity, and cost of using multiple processors. Control and information flow is not flexible and communication between the sensors is not possible.

For example, in Rosenfeld's pyramid architecture [23] and [24], there are layers of processors and each processor gets information just from its neighbors at the bottom layer and passes the information to the layer above. The data flow is unidirectional and feedback is not possible. Thus again, these kinds of solutions optimize the use of multiple processors, but only to a specific, well defined task, in which the order of processing can be prespecified at the design time. As it is stated earlier in this thesis, this kind of a solution is not appropriate for an application with a high level of uncertainty.

2.6 CORBA

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly increasing number of hardware and software products available today [19]. CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 [1] was introduced in 1991 by Object Management Group (OMG). CORBA defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across

[1] OMG Document Number 91.12.1

a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass the object the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In doing so, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In fielding typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and other factors. With an ORB, the protocol is defined through the application interfaces via a single implementation language-independent specification, the IDL. And ORBs provide flexibility. They let programmers choose the most appropriate operating system, execution environment and programming language to use for each component of a system under construction. They allow the integration of existing components. With CORBA, users gain access to information transparently, without knowing what software or hardware platform it resides on or where it is located on. More information about CORBA and its components can be found in Appendix A.

CORBA is a set of de-facto standards for interfaces for object-oriented software architectures and interoperability among these architectures. It is not CORBA's goal to become one software or hardware platform or to set implementation details for these platforms. As a result, CORBA is an appropriate tool to define interfaces for the RAACR's software bus. But there are many issues that are not addressed by CORBA. Among them, the issue of how to utilize CORBA compliant components to build domain specific software architectures is not addressed. One of the goals of this thesis was to investigate how to utilized CORBA standards to build software that reacts to changing requirements through its adaptation mechanisms.

2.7 DSSA: Domain-Specific Software Architecture

Domain-Specific Software Architecture (DSSA) is a project sponsored by the Devense Advanced Research Projects Agency (DARPA). As it is defined in [28], DSSA is a process and infrastructure that supports the development of a *domain model, reference requirements*, and *eference architecture* for a family of applications within a particular problem domain. DSSA aims to support the generation of applications within a particular domain (also known as a *product-line*). Here, *domain* is defined by a set of common problems or functions that applications in that domain can solve.

The goals of DSSA approach are:

- to construct a *Domain Model* that includes the terminology and semantics which characterizes elements, processes and relationships of that Domain,
- to impose standard requirements upon a family of applications (*Reference Requirements*),
- to define and implement the generic *components*[2] which will be the

[2] *Component* is defined as an abstractly characterized unit that can provide capabilities or services to others. Generic components in a reference architecture may be parameterized or otherwise imprecise. However, implementation components are executable and generally have a physical embodiment in software or hardware [13].

construction blocks of *Reference Architecture*,

- to develop a *Reference Architecture*[3] from the Domain Model.A Reference Architecture will be built of implementation objects.

- finally to develop an Application Architecture using the Domain Model, the Reference Requirements and the Reference Architecture of that application.

A client/server architectural style is applicable to DSSA. However, each domain may use several different software architectures, based on their requirements. This brings the vendor flexibility in choosing the software architecture, operating system and even the programming languages. More information on the DARPA project can be found in [14].

While the DSSA project is very closely related to the problems addressed in this thesis, it does not provide ready solutions. For instance, a DSSA solution will not work when architectural standards are not set or when the requirements can not be defined thoroughly. In other words, unexpected changes that happen in the requirements, or unexpected inputs from an uncertain environment that occur during run-time, will cause the DSSA based systems to fail. The solutions that DSSA provides require the involvement of the software designer, who would have to reconfigure the software in order to address the unexpected changes. In this thesis, we investigate a solution which goes beyond the ideas proposed in the DSSA project. We investigate mechanisms that could be invoked automatically

[3] An *Architecture* is defined as a model of an entity in [14]. It enables the functions and flows described in a Domain model to occur. A Reference Architecture is generic. If there are multiple different technologies available for enabling the functions and flows than there can be multiple reference architectures. A Reference Architecture is object oriented while a Domain Model is flow and function oriented.

by the system and would not require the involvement of the software designer.

Chapter 3

Description of the RAACR

3.1 Overview

This chapter gives an informal specification for the Reconfigurable Architecture for Adapting to Changing Requirements (RAACR). Although this specification and the scenarios are specific to the object detection/recognition domain, the proposed system can be easily adapted to other domains.

In Section 3.2, a brief definition of the problem and the goals for defining the RAACR are given. In Section 3.3, the architectural and implementational constraints are discussed. In Section 3.4, the architectural components of the RAACR, abstraction levels (domains) knowledge sources (KSs) and software bus (SB) are described. In Section 3.5 interface definitions of the domains and the communication scheme are defined. In Sections 3.6 adaptation mechanisms (adaptable control and restructurable control schemes) and feedback passing mechanisms are described. Finally, in Section 3.7 extendibility issues are discussed.

3.2 Problem Statement

The subject of this specification is a domain-specific (object/target recognition) software system that incorporates some of the ideas of adaptable and reconfigurable control for dealing with the issue of incomplete specification of requirements. The mechanisms described in this specification are generic mechanisms rather than specific to the domain of object recognition.

3.3 Features of the Architecture

The system consists of multiple layers organized in a hierarchy. Each layer in the hierarchy is related to the layers above and below it through the relation of *abstraction*. Each layer deals with a specific type of data and consists of components which process this data. More detail regarding the structure of layers is in Section 3.4.

The system is object-oriented. The common properties of object-oriented programming, *abstraction*, *inheritance*, *encapsulation* and *polymorphism* are utilized in this architecture. By reducing the interdependency among software components, the reusability and reconfiguribility of the architecture improves [3].

Encapsulation is used to hide the implementation details and the particular data or information at each abstraction level. There are object methods which process the data specific to that abstraction level. These methods are called the Knowledge Sources (KS) since they use their knowledge to process the data. Since the layers of the system are organized according to the abstraction relation, then this hierarchy imposes some constraints on the KSs: they can process only the information at the same abstraction level. Not only does this reduces the amount of complexity of the processing, but it also guarantees that the information that the KSs deal with is semantically in the same category.

The system is adaptable to various inputs, goals and noise characteristics. Therefore, one of the important aspects of the organization and the KSs of the system is their *adaptability*. For this, a *feedback* mechanism is provided. More detail regarding adaptation and feedback mechanism is provided in Sections 3.6.

Multiple KSs may process the same input and produce the same type of output. Since, only one of these outputs will be used later, this brings a competition among the KSs.

A Software Bus (SB) provides the communication channel between the abstraction levels and within the abstraction levels. The flow of information in the RAACR is from lower levels of abstraction to higher, and from higher to lower. KSs at each abstraction level can access the data belonging to the same abstraction level.

The system is "open". It allows addition and removal of the abstraction levels and the KSs at the design time. It decreases the cost of later modifications to the system. The addition or removal of a KS does not change the existing interfaces, neither between the *User* and the SB nor between the existing KSs and the SB.

The RAACR is a CORBA[1] *compliant systems architecture.* More information about CORBA is given in Section A.

[1]The Common Object Request Broker: Architecture and Specification. OMG Document Number 91.12.1 Revision 1.1

3.4 Components of the RAACR

The top-level view of the RAACR is shown in Figure 3.1. The main components of this architecture are *domains*, *software buses*, and *databases*. The external elements are the *user* and *sensors*. The domains are shown as boxes with dashed-line borders. The domains consist of three kinds of components (shown as boxes inside of the domain representation): *data transfer components (DTC)*, *domain scheduler (DS)* and *knowledge sources (KS)*. Knowledge sources are the main working modules of an application. Data transfer components provide the transfer of data between the domain and databases or among the domains. Domain schedulers coordinate the domains; they schedule knowledge sources according to the type or request (data) that needs to be processed, and communicate with other domains.

3.4.1 Domains

RAACR applications run as collections of *domains*. Each domain is a class (in the object oriented sense). The class *domain* is represented in Figure 3.2 using the Rumbaugh representation [25]. The generic attributes of this class are two *object references* (*domain-above*, *domain-below*), i.e., the CORBA compliant addresses of the two domains that are directly above and directly below a given domain. The generic methods of *domain* are: *ScheduleKS*, *TakeData*, *consume, copy, duplicate, marshall*. *TakeData* is a mechanism for transferring results of processing among the domains. *ScheduleKS* is a method of transferring feedback among the domains, as well as a way of requesting processing. The remaining generic methods are required by CORBA.

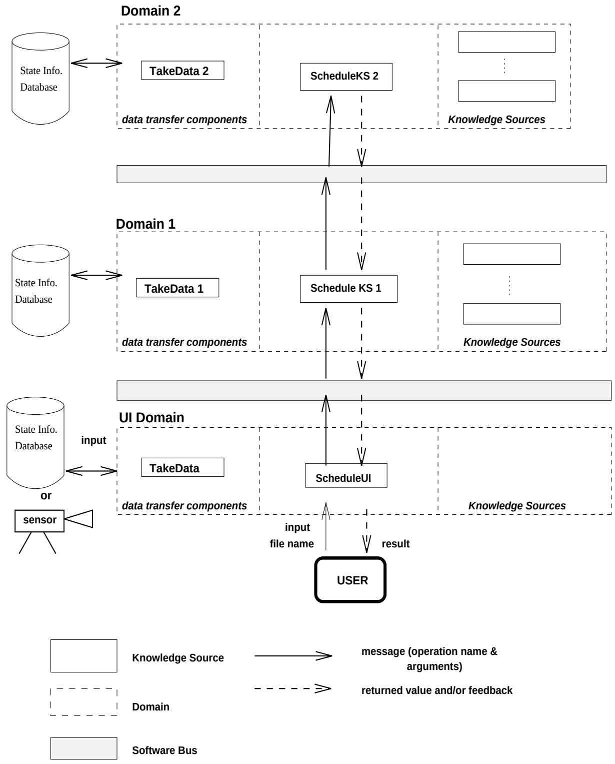

Figure 3.1: **Top-level view of the RAACR.**

Domains are defined by the user. User domains inherit generic attributes and methods provided by the base domain shown in Figure 3.2. Users may add new attributes and operations specific to their application.

Knowledge sources are components of domains; they implement the main functionality of the application. According to the control terminology, we call this functionality *plant*. Each KS must have the following attributes: *feedback*, *model-parameters*, *controller-parameters*, *control-input* and *goal*. Each KS must have the following operations: *calculate-feedback*, *estimate-model*, *design-controller*, *calculate-control*. Knowledge Sources are implemented by the member functions of the the class *domain*.

The RAACR system was implemented on the CORBA compliant SPRING operating system [11, 12] using the C++ programming language. Domains

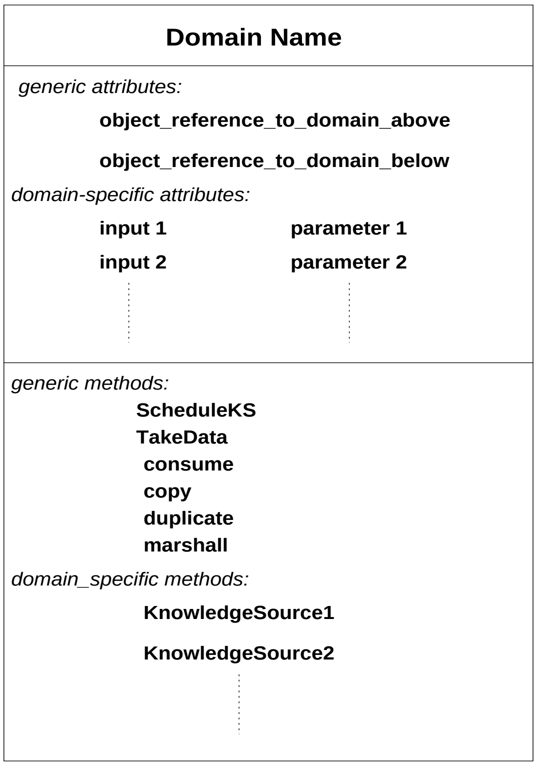

Figure 3.2: **Domain Model for the RAACR**

are implemented as objects. Knowledge sources are methods of domains. Only domain schedulers can communicate with other domains, i.e., with domain schedulers of other domains.

3.4.2 Software Bus

The communication mechanism uses the CORBA standard, as implemented in SPRING. SPRING Client-Server Transactions [27] were used for Inter-Object Communication (see Figure 3.3).

To establish a communications link, first the interface between two objects, called *client* and *server*, must be defined. The CORBA Interface Definition Language (IDL) is used for this purpose. Based upon such a definition, two *stubs* are generated automatically by the system: the *client-side stub* and the *server-side stub*. Also a communications buffer, called *subcontract*, is established. When a client invokes a method of another object (server), the client-side stub calls the server's subcontract and loads arguments into its buffer. The subcontract then invokes the server-side stub, which in turn invokes the server's method.

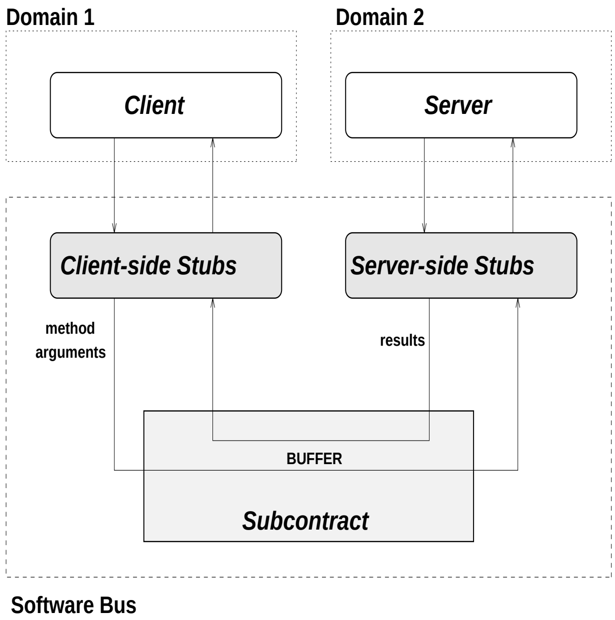

Figure 3.3: **Software Bus**

Message flow between the RAACR domains is shown in Figure 3.4. Message passing among the domains of the RAACR is restricted to the neighbor domains only. Each domain may be either a client, a server or both. For example User Interface domain is a pure client, Histogram domain is a pure server and Image domain is both server and client.

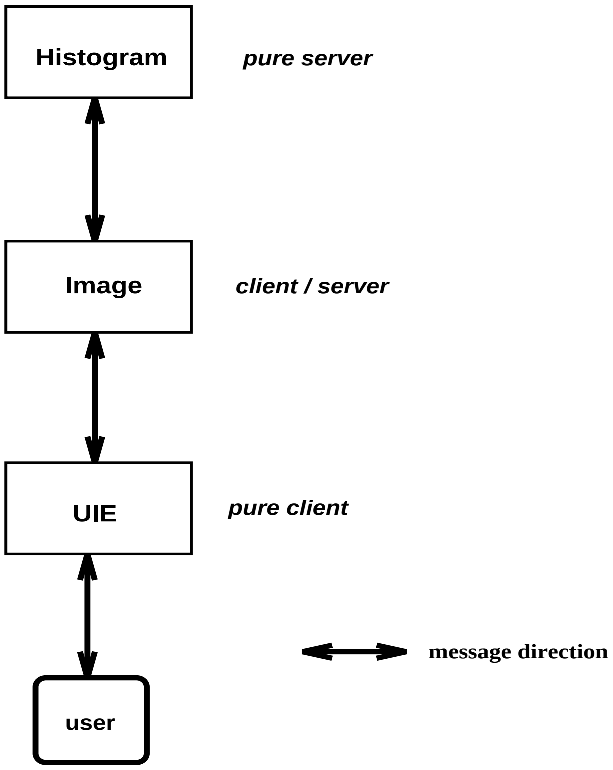

Figure 3.4: **Message flow among the domains of the RAACR.**

3.5 Interfaces among Domains

The interfaces between the various elements of the RAACR are specified using the Interface Definition Language (IDL) defined by CORBA. The syntax and semantics for the IDL conforms to the standards specified by the Object Management Group for IDL C++ Language Mapping Specification [19].

The RAACR consists of multiple levels, or in other words, multiple client/server couples, as shown in Figure 3.4. Only the domain scheduler in each domain has an interface to the next domain (above) and is responsible for the activation of that domain. For instance, as shown in Figure 3.1, *ScheduleKS1* in Domain1 and *ScheduleKS2* in Domain2 are such domain schedulers. Data transfer components of a domain are accessible by the domain scheduler of a domain below it. They get the initial information from the domain below and perform initialization. *TakeData1* in Domain1 and *TakeData2* in Domain2 are data transfer components. Knowledge sources of a domain are not accessible to other domains. Knowledge sources are invoked by the domain scheduler of that domain.

The RAACR shown in Figure 3.1 works as follows. The UIE has an interface (name and argument types of object methods) to the methods *TakeData1* and *ScheduleKS1* in Domain1. First, the UIE calls *Takedata1* by sending a message consisting of the name and the arguments of this method. The message is loaded in the buffer allocated by the software bus. Then, the software bus calls the implementation of *Takedata1* and *ScheduleKS1* methods and passes the arguments from its buffer to these methods. The UIE has no access to the member data of Domain1. *ScheduleKS1* invokes knowledge sources to process the data. After all the knowledge sources complete processing the data, *ScheduleKS1* transfers the processed data to the next domain (Domain2).

Data that is specific to a domain is accessible to all knowledge sources in the domain, but they are hidden from other domains. Messages flow from bottom to top (see Figure 3.1), while feedback or returned values flow from top to bottom. The message passing and domain initialization and activation schemes are the same for all domains.

3.6 Adaptation and Restructuring

The RAACR uses a combination of two control architectures, *adaptive* (see Figure 3.5) and *restructurable* (see Figure 3.6) control architectures.

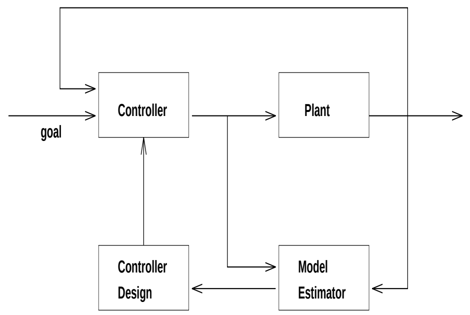

Figure 3.5: **Adaptive Control Architecture**

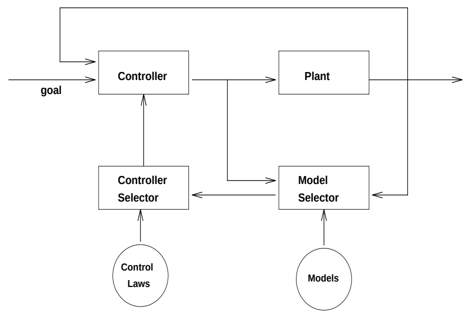

Figure 3.6: **Restructurable Control Architecture**

The main feature of the adaptive control is the ability to deal with the uncertainty in the model parameters of the controlled plant; the exact values of the model parameters do not need to be known at the design time of the adaptive controller [16]. In the RAACR, adaptive control is implemented inside KSs.

According to the adaptive control scheme, $Plant$ receives $control-input$ from $Controller$. These inputs affect the $Plant's$ processing scheme. $Plant's$ output $(feedback)$ is sent to both $Controller$ and $ModelEstimator$. $Controller$ calculates $control-input$ according to its $controllaw$. $Controllaw's$ parameters are updated by $ControllerDesign$ based on the input from $ModelEstimator$. $ModelEstimator$ estimates parameters of the $Plant's$

$model$, based upon both $controlinputs$ and $Plant'soutputs$. Model parameters of the KS are initialized with a predetermined set of values stored in *State Information Database* of each domain.

As it is discussed in [16], adaptive systems are not applicable when control plants change *structurally*, i.e., when adaptation in parameters of either a model or a control law does not provide sufficient performance improvement. As a result, the RAACR utilizes restructurable control, in which the controller *reconfigures* itself when a new situation is identified. In the RAACR, restructurable control is performed by domain schedulers.

Restructuring in the RAACR takes place in two forms: method selection and method invocation sequence. The first restructuring mechanism is to pick the best output of the object methods performing the same operation. For instance, in the case of edge detection application, two of the methods performing edge detection are *SobelEdgeFinder* and *LaplacianEdgeFinder*. Since *SobelEdgeFinder* is a directional edge finder, it detects horizontal and vertical edges more precisely than *LaplacianEdgeFinder*. However, for rotated or round objects, *LaplacianEdgeFinder* performs well since it is a directionless operator. It is the domain scheduler's responsibility to choose the best output among the outputs of object methods performing the same operation with different means.

The second restructuring mechanism is to vary the sequence of the object method invocations (see Figure 3.7). Assume, three operators are implemented by three object methods, A, B and C. If these operators are linear, Associative Law and Commutative Law hold and the order of their invocation does not affect the result. However, if they are nonlinear, to find the correct order of these operators becomes a critical issue. Again, domain scheduler at each domain is responsible for the order of object method invocation.

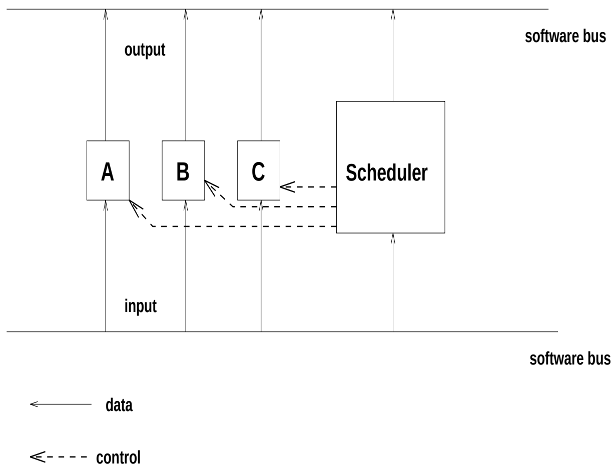

Figure 3.7: **Restructuring is performed by domain schedulers.**

In the RAACR, feedback may be received either as a returned value of an object method or as an exception raised by the object method. Both returned values and exceptions may be complex structures. CORBA defines standard exceptions for the ORB [19]. In the RAACR, domain schedulers provide the mechanism for transferring feedback to KSs within the domain and to domain schedulers of the neighbor domains.

Each feedback type is defined as a structure consisting of an identifier (FeedbackId) and parameters, as shown below.

```
feedback FeedbackID {
         type parameter1;
         type parameter2;
};
```

For example, feedback types can be: `BAD_PARAM` (an invalid parameter is passed), `NO_RESPONSE` (KS does not respond to the request or KS decides not to process the data), `BAD_OPERATION` (an invalid operation is requested).

3.7 Extendibility

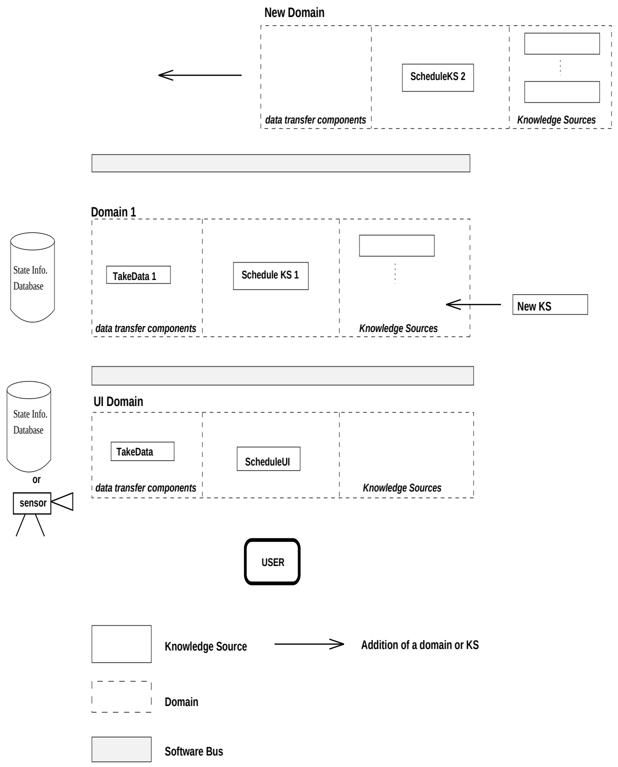

Figure 3.8: **Extendibility of the RAACR.**

The user can add new knowledge sources (KS) to domains of the RAACR off-line. Only interface definitions (stored as *environment files*) and the domain scheduler of the domain, to which a new KS is added, need to be modified. The addition of a new KS is transparent to the rest of the system. To add a new KS to a domain, first, the interface definition (name, parameters and return type) of the KS should be added to the interface definition (environment file) of the domain which contains interface definitions of all KSs. Each KS must have a unique name. The name of the KS, its parameters, the type of the returned value, and exceptions must be added to the environment file. Then the modified environment file should be compiled so that the new KS can be visible to the domain.

The user can add new domains to the RAACR off-line in the same way as adding a new KS to a domain. Depending on whether the domain is inserted between two domains or at the top/bottom, either one or two environment files need to be modified. Then, these environment files are compiled and the ORB creates template source code files to implement the new domain and the stubs to access the new domain. After templates for KSs are obtained, the user provides the source code for the KSs in the templates.

Chapter 4

Description of Experimental Scenario

4.1 Overview

The main goal in designing and implementing a test system was to apply and verify the ideas presented in Chapter 3. In the experimental scenario, the goal of the RAACR is to recognize squares in a binary image. Each binary image in this set should have only one geometric shape. The input to the RAACR is a binary image and the output is a decision, "a square" or "not a square". Other possible geometric shapes like rectangles, triangles or filled circles, can be in the image; in such a case the decision should be "not a square". The algorithm recognizes an object by looking at the number of edges of the object and the angle between the edges of the object. Therefore, the RAACR detects edge points of the geometric object in an image first. Then, it finds the edges and the directions of the edges in order to find the angles between the edges. This two step detection process with two abstraction levels is a good example case for the multi-level (hierarchical) object detection/recognition problem.

As it was required in the specification, the system followed the CORBA standards. The programming language was C++ and the operating system was the CORBA compliant SunSoft's SPRING [1] operating system. The RAACR system was implemented on a Sun Microsystems SPARC-5 workstation.

4.2 Domains and Objects of the Scenario

The domains within the RAACR are as follows (see Figure 4.1):

- User Interface Domain (UI)
- Image Domain
- Histogram Domain

UI: UI is the User Interface Domain. The *UI* takes commands and inputs, as well as feedback, from the command line. It asks the user for the name of the file that contains an input image with a geometric object. It returns the decision made by the RAACR about the shape of the object.

Image Domain: This is the second level in the hierarchical architecture. It is the domain where the initial image data are loaded and edge detection is performed. Once the edges are detected, every edge point is stored in an array. Then, for each edge point the direction of the intensity gradient θ is calculated and a histogram is constructed. This histogram shows the number of edge points $H(\theta)$ for each θ between 0 and 360 degrees.

[1] Spring Research Distribution 1.1 . Copyright (c) 1995 by Sun Microsystems, Inc.

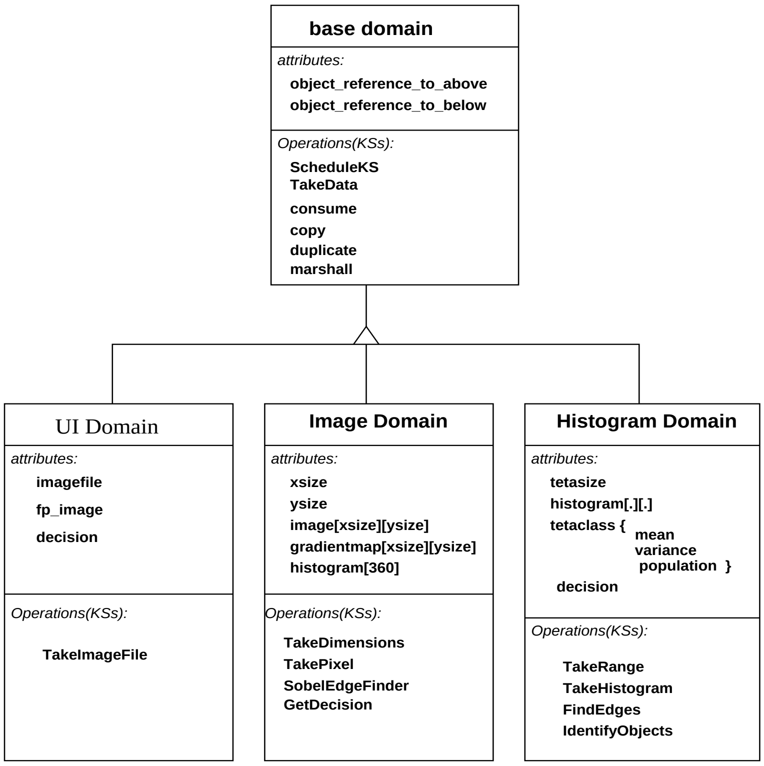

Figure 4.1: **Object Model for the object recognition scenario.**

Histogram Domain: In this domain, the histogram $H(\theta)$ is analysed and edge points are classified according to their θ. The classification method uses a modified MacQueen k-means algorithm. The θ classes are used to determine the number of edges and the angle between the edges. Finally, the shape of the object in the image is determined by using the edge and angle information.

Message flow among the domains was implemented as defined in the RAACR (see Figure 4.2).

4.3 Interface Definitions for the Scenario

The following are the interfaces used by the experimental scenario:

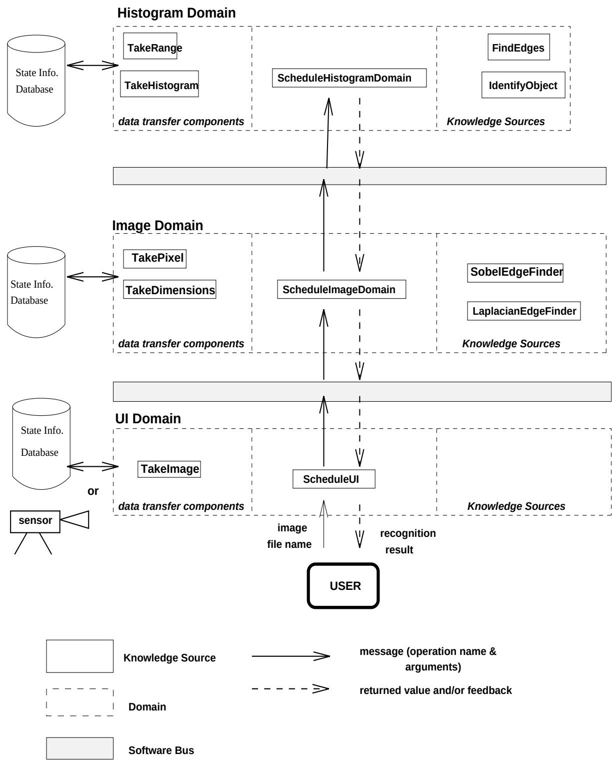

Figure 4.2: **The domains of the experimental system.**

```
interface UI {
     void TakeImage (copy string imagefilename);
     void ScheduleUI();
};

interface Image  {
     void TakeDimensions (copy long xsize,copy long ysize);
     void TakePixel (copy long x, copy long y, copy long value);
     long ScheduleImageDomain ();
};
```

```
interface  Histogram {
     void TakeRange (copy long range);
     void TakeHistogram (copy long teta, copy long number);
     long ScheduleHistogramDomain (copy long edge);
};
```

4.4 Functional Descriptions of the Knowledge Sources

In this section, the functional descriptions of the knowledge sources are given. The format of the references to the knowledge sources is (domain name)::(Knowledge Source name).

4.4.1 Image::TakeDimensions

This KS takes two inputs, width and height of the image, from the UI and initializes the Image domain. Once the object is initialized, memory for a two dimensional integer array, which will hold the input image, is allocated.

4.4.2 Image::TakePixel

This KS takes three inputs: $x\_coordinate, y\_coordinate, pixel\_value$ for each pixel. The UI passes the input image to the Image domain pixel by pixel.

4.4.3 Image::SobelEdgeFinder

The intensity of the image of interest can be represented as a function of image coordinates. The function may have a value of either 0 or 255 at a particular location.

Gradient operators, like Sobel operators are based on local derivatives of the image function. Derivatives are bigger at locations of the image where the image function shows rapid changes and aim to indicate such locations in an image. Sobel operator is used to calculate the resulting value in the output image pixel $f(i, j)$ as a linear combination of brightness in a local neighborhood Ω of the pixel $I(i, j)$ in the input image. The contribution of the pixels in the neighborhood Ω is weighted by coefficients h

$$f(i, j) = \sum_{(m,n)\in\Omega} h(i - m, j - n)I(m, n) \tag{4.1}$$

Equation 4.1 is equivalent to discrete convolution with the kernel h, that is called a **convolution mask**. As a gradient operator, Sobel operator is used to locate abrupt changes in the intensity function, which gives edges in the input image. Edges usually represent object boundaries and the gradient at a particular edge point is perpendicular to the edge. The gradient magnitude $|gradI(x, y)|$ and gradient direction θ can be calculated as

$$|grad\, I(x, y)| = \sqrt{(\frac{\delta I}{\delta x})^2 + (\frac{\delta I}{\delta y})^2} \tag{4.2}$$

$$\theta = \arctan \frac{\delta y}{\delta x} \tag{4.3}$$

where θ is the angle (in radians) from the x-axis to the point (x,y). A digital image is discrete in nature. Therefore the equation 4.2 containing the derivatives must be approximated by differences. The first differences of the image I in the vertical direction (for fixed i) and in the horizontal direction (for fixed j) are given by

$$\Delta_i I(i,j) = I(i+n,j) - I(i-n,j) \tag{4.4}$$

$$\Delta_j I(i,j) = I(i,j+n) - I(i,j-n) \tag{4.5}$$

where n is a small integer, usually 1. Sobel operators represented by the following convolution masks are used to approximate the difference equations in 4.4.

$$h_1 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad h_2 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad h_3 = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

The Sobel operator is often used as a simple detector of horizontality and verticality of edges in which case only masks h_1 and h_2 are used.

Once the Sobel operators are applied and the gradient image $|grad\, I(i,j)|$ containing the edges is obtained, a thresholding procedure is applied to suppress the artifacts. And the final edge image $f(i,j)$ is

$$f(i,j) = \begin{cases} 0 & if\ |grad\, I(i,j)| < threshold \\ 1 & otherwise\ (edge\ point) \end{cases}$$

An histogram of the edge points can be constructed according to their direction θ obtained in equation 4.3.

4.4.4 Histogram::TakeRange

This KS takes one input which is the range of the histogram and allocates memory for a one dimensional array which will hold the histogram created in the Image Domain.

4.4.5 Histogram::TakeHistogram

This KS takes two inputs, θ the gradient direction and $H(\theta)$ the number of edge pixels with a gradient direction θ. Each time, this KS is called, it sets one element of the Histogram array.

4.4.6 Histogram::FindEdges

For a perfect polygon with n edges (or sides), there are n θ values that a direction of an edge point may take. However, in practical situations, *n classes* of θ values are found instead of n certain values. Once, the θ values are classified and the number and the mean of different classes are determined, the RAACR can decide the shape of the object. In typical classification problems, the number of classes is known beforehand. However, in the square detection problem, the number of classes, which represent the number of edges of the object is unknown. Therefore, the RAACR utilizes a statistical classifier, to classify the θ values obtained at the previous steps and learn the number of classes or the number of edges.

A *statistical classifier* is a device with m inputs and one output. Each input is used to enter the information about one of m features that are measured from a classified object. In the square detection problem, objects are the edge points and the only feature is the direction θ. An R class classifier, where R is the number of classes known beforehand, generates one of R symbols $w_1, w_2, \ldots, w_R$ as an output, and the user interprets this output as a decision about the class of the processed object. The generated symbols are **class identifiers**. The function $d(x) = w_r$ describes the relation between the classifier inputs and the output, and is called the **decision rule**. Assume that R points are defined in the feature space $v_1, v_2, \ldots, v_R$ that represents the **exemplars** or the sample patterns of classes $w_1, w_2, \ldots, w_R$. A *minimum distance classifier* classifies a pattern x into the class w_r to whose exemplar it is closest,

$$d(x) = w_r \Longleftrightarrow |v_r - x| = \min_{s=1,\ldots,R} |v_s - x|. \tag{4.6}$$

In this KS, a variation of the *MacQueen k-means* classification method [18] was used. Since the MacQueen k-means method needs the number of classes as input (which is not known initially), a modification was made to the *MacQueen k-means* classification method so that it determines the number of classes dynamically as it is explained in [7].

In our algorithm, the classification rule consists of three parts. The algorithm first selects θ_e (*exemplars*) to represent initial classes and determines the number of these initial classes by using the following procedure. The algorithm selects the θ with maximum $H(\theta)$ as an exemplar and surrounds it with a box of width δ which is pre-determined and re-adjusted during the detection process. Then the algorithm selects another exemplar (θ with the highest $H(\theta)$) from the θ values outside of the box. This procedure is iterated until all the θ values are inside of at least one box. The result is a list of initial exemplars and the initial number of classes k.

In the next step, the classification algorithm assigns each edge point (except for those selected as exemplars) to one of the classes using the classification decision rules $d_i(p) = w_r$ shown below:

$$d_1(p) = w_r \quad \Longleftrightarrow \quad \|v_r - p\| = \min_{s=1,\ldots,k} \|v_s - p\|. \tag{4.7}$$

$$d_2(p) = w_r \quad \Longleftrightarrow \quad \|v_r - p\| \leq max\_distance. \tag{4.8}$$

where p represents a pixel, w_r represents the $r-th$ class, $\|v_r - p\|$ represents the Euclidian distance between the center v_r of the class w_r and the pixel p.

After a new edge point p is added to the class i the centroid v_i for this class is updated according to the following rule:

$$v_i = \frac{1}{k_i + 1}(k_i v_i + p) \tag{4.9}$$

$$k_i = k_i + 1 \tag{4.10}$$

where k_i is the number of edge points included in class i, v_i is the center of the class and p is the newly added edge points. These centroids are used as new exemplars in the next pass. The process is iterated until no re-assignments of edge points to classes are made in one pass.

Additionally, for each class the algorithm computes the mean θ and the number of edge points (*population*).

4.4.7 Histogram::IdentifyObject

This KS gets the information about the θ classes as an input. It first sorts the θ classes according to their pixel population. The population of each θ classes are compared with a minimum population threshold and this KS determines whether these classes qualify as an edge or not. The minimum population threshold is initialized a certain constant at the beginning. Then, the KS counts the θ classes which qualify as an edge. If the edge number is four, the KS computes the angles between the neighbor edges to check whether they are all right angles or not. If they are all right angles, the KS compares the lengths of all four edges.

X_i is defined as a random variable for the gradient angle calculated at edge pixels of the $i-th$ edge. The random variable X_i is assumed to have a normal distribution with, $N(X_i; \mu_i, \sigma_i)$. $\overline{X_i}$ is the mean of a sample of edge pixels belong to $i-th$ edge and an unbiased estimator of μ_i which gives the direction of $i-th$ edge. s_i^2 is the variance of the same sample.

X_i and X_{i-1} are normally distributed random variables related to two neighbor edges of the geometric object and the difference between the directions of these edges $\mu_i - \mu_{i-1}$ are estimated .

If the variances of X_i and X_{i-1} are unknown and are not same, then the **Smith_Satterthwaite** test can be employed to estimate the difference between the directions of neighbor edges.

The following hypothesis test is applied for all neighbor edge couples:

$$H_0: \quad \mu_i - \mu_{i-1} = 90 \tag{4.11}$$

$$H_1: \quad \mu_i - \mu_{i-1} \neq 90 \tag{4.12}$$

with a test statistic

$$T = \frac{X_i - X_{i-1} - 90}{\sqrt{\frac{s_i^2}{n_i} + \frac{s_{i-1}^2}{n_{i-1}}}}$$

Given H_0, the distribution of T is approximated by a t distribution with a degrees of freedom

$$v = \frac{(s_i^2/n_i + s_{i-1}^2/n_{i-1})^2}{\frac{(s_i^2/n_i)^2}{n_i-1} + \frac{(s_{i-1}^2/n_{i-1})^2}{n_{i-1}-1}}$$

Since it is not always an integer, v must be rounded up. A two-tailed hypothesis test for $\mu_i - \mu_{i-1}$ with a *confidence coefficient* α is of the form

$$LL_i \leq T \leq UL_i$$

where

$$UL_i = t_{\alpha/2,v} \tag{4.13}$$

$$LL_i = -t_{\alpha/2,v} \tag{4.14}$$

The decision rule is as follow:

$$\text{decision} : \begin{cases} H_0 & \text{if } UL_i \geq T \geq LL_i \\ H_1 & \text{otherwise} \end{cases}$$

If all four angles are within the α confidence interval of 90 degrees and the lengths of all edges are equal then the KS decides that the object is a "square".

4.5 Adaptation Mechanism for IdentifyObject

In order to present the adaptation and the restructuring mechanisms we need to view each knowledge source as a small system with an autonomous adaptive controller. Thus each knowledge source must contain all the components, as shown in Figure 3.5: *Plant, Model Estimator, Controller Designer*, and *Controller*. Below we describe each of these components for the knowledge source *IdentifyObject* (see Figure 4.3).

The main function of *Plant* is to input one image $image_t$ at a time, indexed by the image frame number t and to produce a decision, *square* or *non-square*. *Plant* makes this decision based upon a confidence constant α which is a control input from *Controller*. This decision rule described by Eq. 4.11 in Section 4.4.7.

Additionally, *Plant* outputs some intermediate results to *Model Estimator*. This functionality could be implemented directly in *Model Estimator*, but we included it in *Plant* for the sake of efficiency of computation. This additional output includes $\overline{X_i}$ and s_i, where $\overline{X_i}$ is the value of the gradient direction for edge i, and s_i is its estimated variance. The details of calculating these values were presented in Section 4.4.7.

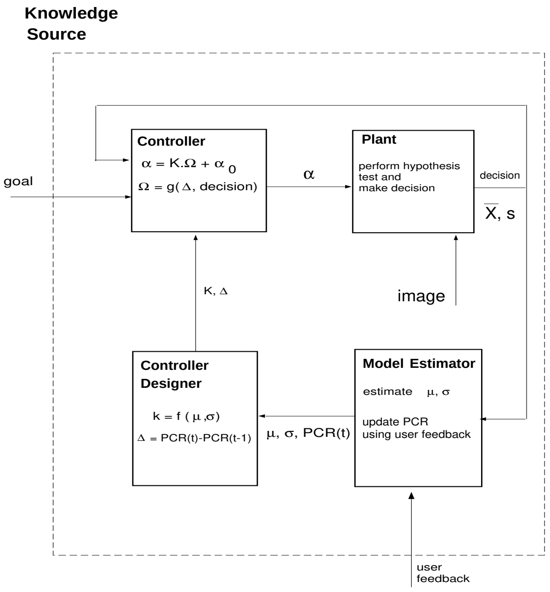

Figure 4.3: **Adaptation Mechanism for IdentifyObject.**

Model Estimator estimates the model of the plant. The model is probabilistic. It is updated incrementally after receiving an input from the plant ($image_t$) and decision d, and the correct decision from the user. The model is represented by a normal distribution $N(D_i; \mu_i, \sigma_i)$, where D_i is the angle between two consecutive edges i and $i-1$

$$D_i = \overline{X_i} - \overline{X_{i-1}}.$$

In the adaptation mode, after each decision, $ModelEstimator$ receives user's $feedback$. This feedback's value is 1 if $Plant$ made a correct decision, and 0 otherwise. In response to feedback, $ModelEstimator$ updates $PCR(t)$ according to the feedback update equation 4.15

$$PCR(t) = \frac{PCR(t-1) \times (t-1) + feedback}{t}. \tag{4.15}$$

ModelEstimator updates the estimates for μ_i and σ_i after obtaining new readings of $\overline{X_i}$ and s_i^2 from *Plant* according to the following rule:

$$E(D_i) = \overline{X_i} - \overline{X_{i-1}} \tag{4.16}$$

$$s(D_i) = \sqrt{\frac{s_i^2}{n_i} + \frac{s_{i-1}^2}{n_{i-1}}}, \tag{4.17}$$

where n_i is the number of edge classes for $image_t$. The updated estimates for μ_i, σ_i and $PCR(t)$ are passed to the *Controller Designer*.

Controller Designer updates the gain K and parameter Δ of the control law according to the following equations:

$$K = const. \times \sigma_i$$

$$\Delta = P_{CR}(t) - P_{CR}(t-1)$$

Controller receives as input: *decision* from *Plant*, Δ and K from *Controller Designer* and *goal*. The goal is the decision that *Plant* is supposed to make, i.e., a correct decision. *Controller* applies its control law to calculate the control input:

$$\alpha(t) = K \times \Omega + \alpha(t-1), \tag{4.18}$$

where Ω is calculated according to the following rule:

$$\Omega = \begin{cases} 0 & \text{if } \Delta > 0 \\ 1 & \text{if } \Delta < 0 \text{ and } decision = square \\ -1 & \text{if } \Delta \leq 0 \text{ and } decision = non-square \end{cases}$$

The new value of α is passed to the plant as the new control input.

4.5.1 Restructuring Mechanism for the Image Domain

The restructuring mechanism of restructurable control (see Figure 3.6) is implemented in the domain schedulers (DS). In our application the restructuring mechanism was implemented in the *Image* domain. *Domain Scheduler* makes a decision on which of the knowledge sources to schedule next. It bases its decision upon the feedback it receives from the knowledge sources. In our application, the *Image* domain scheduler receives feedback from *IdentifyObject*. The feedback has the value of 1 if the knowledge source (*Identify Object*) is satisfied with the input provided to it from either *SobelEdgeFinder* or *LaplacianEdgeFinder*, and 0 otherwise.

The feedback is calculation is based on the number of edges. *IdentifyObject* expects a geometric object to have close to 4 edges. If there is an extreme case, (number of edges > 5 or < 3), that means the edge detector does not work well. Then, *IdentifyObject* sends a feedback with a value of 0. In response to such a feedback, the domain scheduler schedules a different knowledge source.

The IDs and addresses of KSs are kept in an array. DS searches the array to find the address of the KS using the KS ID as an index. After DS finds the address, it invokes the KS. Then, DS waits for feedback from the domain above (*Histogram* domain). If the feedback is 0, then DS of the *Image* domain invokes the next KS in the array. If the feedback is 1, it saves the ID of the KS in the database, so that it can be used next time around.

Chapter 5

Measurable Changes in Requirements for the RAACR

5.1 Measuring Requirements Changes

A *requirement* is defined as a statement of a capability that a system must supply or a quality that a system must possess in order to solve a problem or achieve a stated objective (IEEE 1990) [21]. *Volatility* is the term that refers to **change** in a software system product over time [20]. Metrics which characterize the volatility in products such as requirements and designs are used in the monitoring and the stabilization process for developing the product.

The four requirements volatility metrics defined in [4] are:

- *Total Requirements Volatility (TRV)*: tracks the total number of changes in the requirements.
- *Requirements Change Reason (RCR)* : tracks the reasons for the changes in the requirements.

- *Requirements annotated as Potentially Volatile (RPV)* : tracks the number of requirements annotated as “likely to change”.
- *System Characteristics Volatility (SCV)*: tracks the number of changes to requirements that are assigned to each pre-defined category of requirements.

Other studies related to the requirements metrics can be found in [5], [9], [26] and [22].

In this thesis, we chose the Total Requirements Volatility (TRV) metric [4] to measure changes in requirements. According to the methodology described in [4], changes in the requirements must be simplified, divided and enumerated as independent countable pieces. The project life time[1] should be divided into equal time intervals indexed by p and measurements should be repeated for each interval.

Then three measures of change, $RA(s,p)$, $RD(s,p)$ and $RM(s,p)$, need to be computed for each time period p and each specification $S(s,p)$. $RA(s,p)$ counts requirements that were added, $RD(s,p)$ counts requirements that were deleted, and $RM(s,p)$ counts requirements that were modified in the specification s during the time period p. These three measures are calculated according to the following rule:

- If a new requirement is added, this change counts for 1.
- If a new requirement is added and then deleted within the same interval then no change is counted.
- If an existing requirement is deleted, this change counts for 1.

[1] Project life time is the time period from the starting of writing the Requirements Specification for the system, till the last test for the system is completed. In our case, this is the duration of experiments.

- If an existing requirement is modified, this change counts for 1.
- If an existing requirement is split into k sub requirements, this change counts for $k + 1$.
- If k requirements are merged into one requirement, this change counts for $k - 1$.

The Total Requirements Volatility measure, TRV_{cum} is the sum of the three measures. The three measures and the Total Requirements Volatility measure can be formalized by the following formulas:

$$TRV_{cum} = RA_{cum} + RD_{cum} + RV_{cum} \tag{5.1}$$

$$RA_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RA(s,p) \tag{5.2}$$

$$RD_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RD(s,p) \tag{5.3}$$

$$RM_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RM(s,p) \tag{5.4}$$

5.2 Requirements for the Experimental Scenario

In order to measure the changes in the requirements of the experimental scenario, the requirements were simplified, divided and enumerated as independent, countable pieces. Only the requirements related to the square detection application were included in this requirements change analysis. The requirements related to the architectural constraints were not considered for the sake of simplicity. The requirements were collected under four categories, Goal, Analysis/Methods Requirements, I/O, and Functional Requirements. Each requirement is given an ID which is a combination of letters and numbers, where letters indicate its category (G for Goal or M for Analysis/Methods Requirements).

Goal Requirements:

1. (G_1) The RAACR shall detect squares in intensity images.

Analysis/Methods Requirements:

2. (M_1) Object-Oriented Analysis (OOA) shall be used as an analysis method.
3. (M_2) Data abstraction shall be provided.
4. (M_3) Information hiding (encapsulation) shall be provided.
5. (M_4) Inheritance shall be utilized.

I/O Requirements:

6. (IO_1) The RAACR shall accept images as input.
7. (IO_2)The RAACR shall return a decision "square" if it detects a square, "not-square" otherwise.

Image Requirements:

8. (IM_1) Images shall be read from the memory (hard drive).
9. (IM_2) Images shall be of size less than 100 by 100 pixels.

10. (IM_3) Images shall be in `integer` format.

11. (IM_4) Images shall be binary. Possible values for each image pixel are 0 (black) through 255 (white).

12. (IM_5) An image shall contain only one geometric shape.

13. (IM_6) An image may contain either a square or a filled circle.

14. (IM_7) Edges of the geometric shapes shall be aligned with the borders of the image.

15. (IM_8) Images shall be created by a drawing program and shall be noise free.

Functional Requirements:

16. (F_1) The number of edges of an object shall be used to determine the geometric shape of the object.

17. (F_2) The angle between two edges of an object shall be used to determine the geometric shape of the object.

18. (F_3) Edges of an object in the image shall be detected.

19. (F_4) The direction of intensity gradient θ at each edge point shall be calculated.

20. (F_5) An histogram of the edge points $H(\theta)$ shall be constructed. $H(\theta)$ will show the distribution of edge points according to their gradient directions θ.

21. (F_6) $H(\theta)$ should be used for classifying edge points into $\theta - classes$ of points with similar gradient directions. These classes should be

interpreted as edge objects. The number of individual $\theta - classes$ shall give the number of edges of the object.

22. (F_7) The exemplars of each $\theta - class$ shall be calculated as in Equation 4.9.

23. (F_8) A statistical classifier shall be used to classify the edge points and thus determine the $\theta - classes$.

24. (F_9) Edge detection shall be realized by using Sobel operators.

25. (F_{10}) Sobel operators in vertical and horizontal direction shall be used.

26. (F_{11}) The MacQueen k-means method shall be used as a minimum distance classifier.

5.3 Requirement Dependency

Requirements are never independent. Typically, one requirement is related to one or more other requirements. This fact has implications for measuring changes in requirements. A change in one of the requirements usually affects other requirements, and therefore, to measure requirements, one needs to trace the effects of a change in one requirement to other requirements. To simplify the measurement process, we proposed and used in this thesis the concepts of the *Requirements Traceability Matrix (RTM)* and a *Requirements Graph (RG)*.

Consequently, after the requirements were specified and grouped under four categories (Goal, I/O, Image, Functional), a requirements traceability matrix and a requirements graph were constructed (See Figure 5.1 and

Figure 5.2). Both the RTM and the RG represent the parent-child relationships among specifications. In the matrix, the parent-child relation is represented by an "X" in an appropriate field. In the graph, the relation is represented by arrows, with the arrow head pointing to a child. A matrix representation of relations among the requirements is computationally efficient if an automated requirements traceability tool is implemented. However, a requirements graph is more useful for demonstrative purposes because it is easy to trace visually.

The parent-child relation is used to reason about the effects of change in requirements. For example, at the beginning of our experiments three of the initial requirements were: a) Number of edges shall be used to identify the geometric object, b) Edge detectors shall be used, c) Sobel Edge Detector shall be used. For this example, (a) is related to (b) and (b) is related to (c). If the user decided not to use the number of edges as an identification tool and deleted (a) from the requirements, then this would affect the requirements (b) and (c); they become unnecessary and should be removed. The overall effect of deleting (a) must account for the deletion of (b) and (c) as well.

As mentioned earlier, changes in the requirements can be of three kinds: addition, deletion or modification of an existing requirement. The relation between a parent requirement and a child and the effect it has on changes in requirements can be summarized by the following rules:

- A parent requirement may have more than one child requirement.
- A child requirement may have more than one parent requirement.
- A modification in a parent requirement affects all children requirements.

- A modification in a child requirement does not affect its parent or sibling requirements.

- An addition of a requirement does not affect the parent or the siblings of the newly added requirement.

- A deletion of a requirement affects all of its children requirements.

Children

Parents	G1	M1	M2	M3	M4	IO1	IO2	IM1	IM2	IM3	IM4	IM5	IM6	IM7	IM8	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
G1		X				X	X									X	X									
M1			X	X	X																					
M2																										
M3																										
M4																										
IO1								X	X	X		X	X		X											
IO2																										
IM1																										
IM2																										
IM3											X															
IM4																										
IM5																										
IM6														X												
IM7																										
IM8																										
F1																		X		X	X		X			
F2																			X	X		X	X			
F3																								X		
F4																										
F5																					X					
F6																						X				
F7																							X			
F8																										X
F9																									X	
F10																										

Figure 5.1: **Requirements Traceability Matrix (RTM) for the RAACR.**

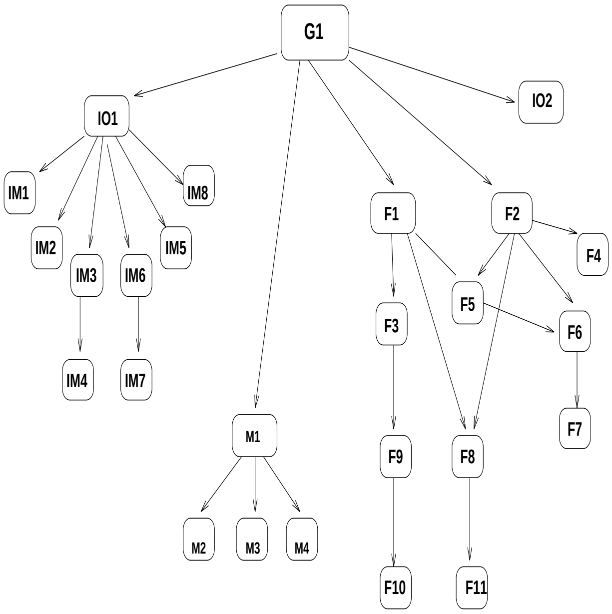

Figure 5.2: **Requirements Graph (RG) for the RAACR**

Chapter 6

Experimental Results

6.1 Incremental Changes

The main idea of these experiments was that the system designed to fulfill a set of software requirements should be adaptable to new more demanding requirements. Therefore, in the first step of our experiments, we tested the system on a set of test data (images) that were within the bounds of the initial specifications. Then, we generated three more sets of images of progressing complexity as measured by the Total Requirements Volatility measure. The *probability of correct recognition (PCR)* was used to measure the performance of the RAACR. Bar charts showing the PCR versus the changes in the requirements were used to show the system's ability to adapt to unexpected changes at run-time.

The four types of input images are specified below.

Set 1 Binary images containing either a square or a filled circle. Images are perfect images, i.e., there is no noise in the data. Edges of the squares are aligned with the frame of the image.

Set 2 Binary images containing one of the following objects: a square, a filled circle, a rectangle, a triangle. Images do not contain noise. Edges of squares, rectangles and the base of a triangle are aligned with the frame of the image.

Set 3 In addition to the images as above, squares are rotated by a randomly selected angle.

Set 4 Shot and Gaussian noise is added to the images of Set 3.

These sets of images are summarized in Table 6.1. Five of the ten images in each set included a square. All geometric objects had different size and different center coordinates. Each new image was assigned to a new time interval; this resulted in four time intervals.

Set: p	*Object type*				
	Square		*Circle*	*Triangle*	*Rectangle*
	aligned	*rotated*			
1	5	-	5	-	-
2	3	-	-	3	4
3	-	5	1	2	2
4	1	5	1	1	2

Table 6.1: **Classification of input images according to the object types included.**

According to the methodology described in Chapter 5, the changes in the requirements have been enumerated in the following way.

1. $S(1,1)$ An image shall contain either a square or a filled circle.

2. $S(1,2)$ Edges of a square shall be aligned with the image frame.

3. $S(1,3)$ Pixel values shall be binary.

4. $S(1,4)$ Images shall be noise free.

5. $S(2,1)$ An image can contain a rectangle.

6. $S(2,2)$ An image can contain a triangle.

7. $S(3,1)$ Squares can be rotated by any angle with respect to the image frame.

8. $S(3,2)$ Diagonal Sobel operators need to be used for detecting edges.

9. $S(4,1)$ Images can contain Gaussian noise.

10. $S(4,2)$ Images can contain shot noise.

11. $S(4,3)$ Images can be of the gray scale type.

These changes in requirements, along with the changes resulting from the dependency relation represented by the requirements traceability matrix, RTM, and an associated requirements graph RG shown in Chapter 5, as well the associated values of the measures RA, RD, RM and TRV are shown in Table 6.3.

Set:	*Added requirements*	*Affected requirements*
1(base)	$S(1,1)$,$S(1,2)$,$S(1,3)$, $S(1,4)$	-
2	$S(2,1)$,$S(2,2)$,	$S(1,1)$(modified)
3	$S(3,1)$,$S(3,2)$	$S(1,2)$(deleted)
4	$S(4,1)$,$S(4,2)$,$S(4,3)$	$S(1,3)$(deleted),$S(1,4)$(deleted)

Table 6.2: **Requirements changes for each image set.**

6.2 Probability of Correct Recognition (PCR)

To evaluate the degree of degradation of the system performance that results from changes in the requirements and of the improvement due to the

Set:	RA	RD	RM	TRV
1(base)	0	0	0	0
2	3	0	1	4
3	2	1	0	3
4	3	2	0	5

Table 6.3: TRV **values for each image set.**

adaptability of the system, we use the probability of correct recognition (PCR) of the object. This probability was calculated as:

$$PCR = \frac{q+n}{N} \tag{6.1}$$

where q is the number of squares recognized correctly by the RAACR, n is the number of non-square objects (filled circles, triangles, rectangles) which the RAACR correctly recognized as non-squares, and N is the number of all input images in a set.

6.3 Comparison of Adaptable and Non-Adaptable Versions of the System

To evaluate the efficiency of the adaptation mechanism of the RAACR, we had to compare the response of the RAACR to the changes in the requirements with the response of a system without the adaptation mechanism. For this purpose we implemented a system that contained the basic functional components (knowledge sources), i.e., `SobelEdgeFinder`, `CreateHistogram`, `FindEdges` and `IdentifyObject`. These components were connected sequentially as shown in Figure 6.1. The adaptation mechanisms of the knowledge sources were turned off. The same sets of input images were input to this non-adaptive system and the probability of correct recognition was calculated. These images were input in the same order as for the adaptive system.

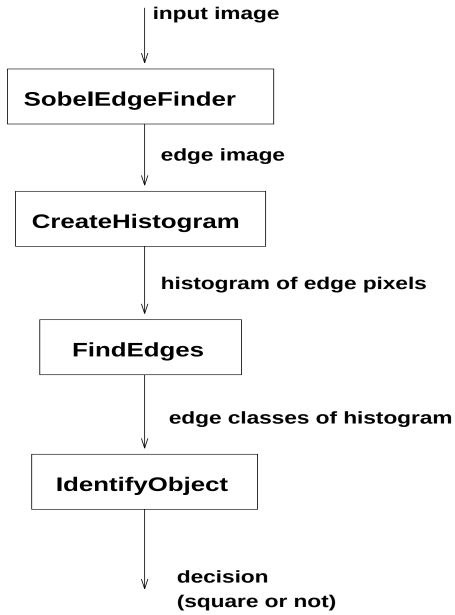

Figure 6.1: **Non-adaptive object recognition system.**

The results of these experiments are shown in Tables 6.4, 6.5, 6.7 and 6.7. Table 6.4 and Table 6.5 show the results of object recognition for the adaptive system. Table 6.6 and Table 6.7 show the results for the non-adaptive system.

The Probability of Correct Recognition was calculated for each TRV. In addition to this, two probabilities were calculated: (1) probability of correctly recognizing squares, and (2) probability of correctly recognizing non-squares. These results are shown in Figure 6.2 (for the RAACR, i.e., with the system with the adaptation mechanism) and in Figure 6.3) (for the non-adaptive system). Finally, in Figure 6.4, the probabilities of correct recognition for the two cases were compared.

Set 1		*Set 2*	
Object:	Decision:	Object:	Decision:
square	sq	rectangle	non-sq
square	sq	rectangle	non-sq
circle	sq	rectangle	sq
circle	non-sq	triangle	non-sq
circle	non-sq	triangle	non-sq
circle	non-sq	square	sq
square	sq	triangle	non-sq
square	sq	square	sq
square	sq	rectangle	non-sq
circle	non-sq	square	sq

Table 6.4: **Recognition results for the RAACR (Sets 1 and 2).**

Set 3		*Set 4*	
Object:	Decision:	Object:	Decision:
rot. square	non-sq	(noise) square	sq
rot. square	sq	(noise) rot. square	non-sq
rot. square	sq	(noise) rot. square	sq
rot. rectangle	non-sq	(noise) rot. square	sq
triangle	non-sq	triangle	non-sq
triangle	non-sq	(noise) rect.	non-sq
rot. square	sq	(noise) rot. square	sq
rectangle	non-sq	(noise) circle	sq
square	sq	rectangle	non-sq
circle	non-sq	(noise) rot. square	sq

Table 6.5: **Recognition results for the RAACR (Sets 3 and 4).**

Set 1		*Set 2*	
Object:	Decision:	Object:	Decision:
square	sq	rectangle	non-sq
square	sq	rectangle	non-sq
circle	sq	rectangle	sq
circle	non-sq	triangle	non-sq
circle	non-sq	triangle	non-sq
circle	non-sq	square	sq
square	sq	triangle	non-sq
square	sq	square	sq
square	sq	rectangle	non-sq
circle	non-sq	square	sq

Table 6.6: **Recognition results for the non-adaptive system (Sets 1 and 2).**

Set 3		*Set 4*	
Object:	Decision:	Object:	Decision:
rot. square	non-sq	(noise) square	non-sq
rot. square	non-sq	(noise) rot.square	non-sq
rot. square	non-sq	(noise) rot. square	non-sq
rot. rectangle	non-sq	(noise) rot. square	non-sq
triangle	non-sq	triangle	non-sq
triangle	non-sq	(noise) rect.	sq
rot. square	non-sq	(noise) rot. square	non-sq
rectangle	non-sq	(noise) circle	sq
rot. square	sq	rectangle	sq
circle	non-sq	(noise) rot. square	non-sq

Table 6.7: **Recognition results for the non-adaptive system.**

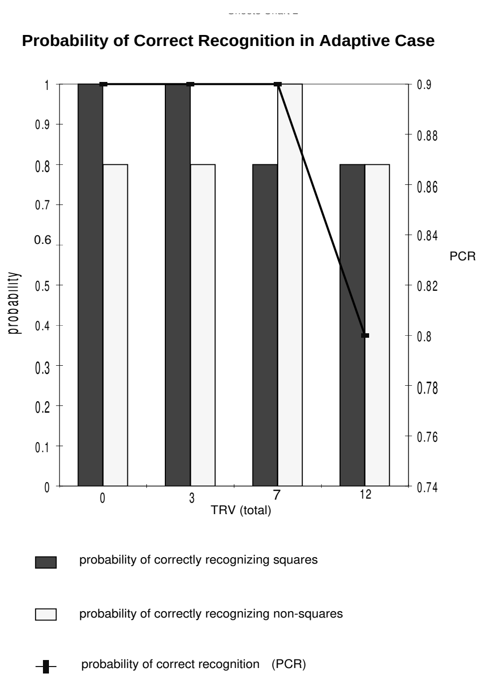

Figure 6.2: **Probability of Correct Recognition for the RAACR**

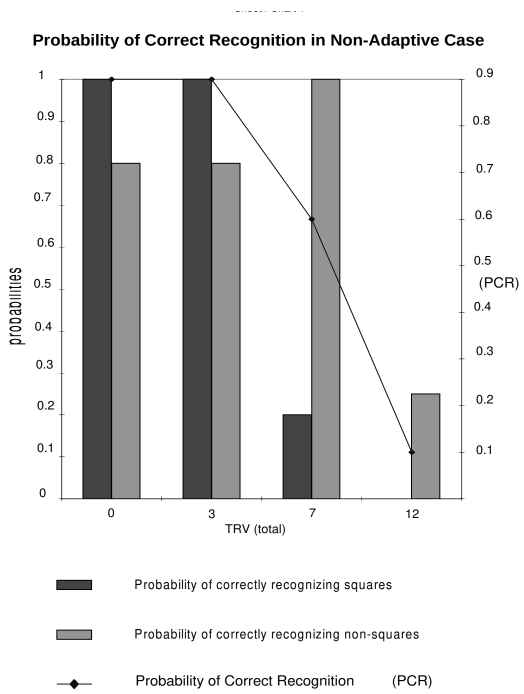

Figure 6.3: **Probability of Correct Recognition for Non-Adaptive System**

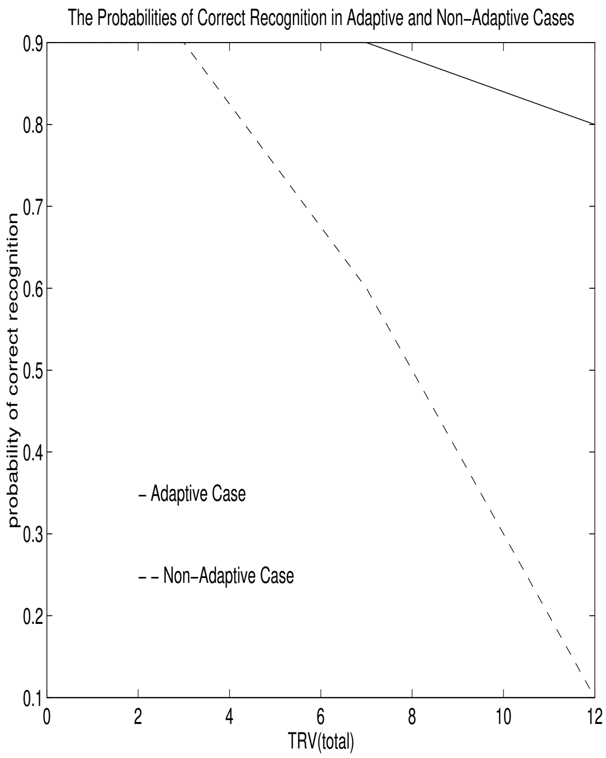

Figure 6.4: **Comparison of the Probabilities of Correct Recognition of Adaptive and Non-Adaptive Systems**

Chapter 7

Conclusions

In summary, we achieved the following results.

1. We identified appropriate adaptive processing mechanisms known in the control literature and transplanted these mechanisms into the software engineering domain.

2. We proposed an architecture for building software systems that are adaptable to changes in the software requirements. The main features of this architecture are:

    (a) Organization of tasks performing similar functions into domains, allowing in this way restructuring of the processing through either selecting one of the processing components or scheduling processing components in various orders.

    (b) Identification of methods for processing feedback. The feedback processing mechanism consists of two functions: calculation of feedback and adaptation of processing parameters.

(c) Hierarchical organization of the processing components as a way of dealing with the complexity of processing.

(d) Generic, standard based (CORBA) communication mechanism among domains.

3. We implemented a system according to the principles described above. We tested the system experimentally.

4. We evaluated quantitatively the quality of the adaptation mechanism in response to changes in software requirements. For this purpose, we used the Total Requirements Volatility measure (to measure changes in the inputs to the system) and the probability of correct recognition (to measure the output of the system). We compared the performance of the adaptable system against the performance of a non-adaptable system. The testing results clearly indicate that the performance of the system with the adaptation mechanism is much better than the performance of the system without the adaptation mechanism. While the performance of the system without adaptation drops dramatically while different classes of inputs are provided, the performance of the adaptive system degrades by only 20%.

While the results of this experiment show that it is possible to build systems that possess the characteristic of on-line adaptation to unexpected changes in software requirements, it should be clear that this is just a small step in the right direction. To achieve the level of knowledge that would allow for engineering of such systems, more research is necessary. This includes such issues as: generic feedback calculation mechanisms, generic adaptation mechanisms, efficient communication mechanisms, support for measuring changes in requirements, and many others.

We have encountered a number of problems during this investigation. Some of these problems are listed below.

1. the C++ compiler provided by SpringOS does not support “math.h” resource file. Therefore, we had to implement simple functions like $sin(x)$ and $\sqrt{x}$.

2. The C++ debugger of SpringOS does not support multi-threaded applications. Therefore, the effort expanded on coding and debugging of the RAACR system was much higher than expected and also much higher than it would have been, given the right software engineering tools.

3. Due to constraints imposed by the SpringOS operating system, we gave up on some of the ideas that we wanted to test initially. For instance, we intended to implement the decision mechanism on whether and when to invoke a knowledge source within the knowledge sources. In such a case, with an autonomy on whether to process the data or not, the knowledge sources would be implemented as *software agents*. Due to the early mentioned constraints imposed by SpringOS, the (centralized) invocation mechanism is implemented within domain controllers.

Appendix A

The Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA), as defined by the Object Management Group (OMG), is a self-contained response to the Request For Proposals (RFP) issued by the Object Request Broker (ORB) Task Force of the OMG. CORBA defines a framework for different ORB implementations to provide common ORB services and interfaces to support portable clients and implementations of objects.

This section describes a brief summary for the ORB architecture and interface description.

A.1 Object Model

The concrete object model underlies the CORBA and provides an organized presentation of object concepts. An object system is a collection of objects that isolates the requesters of services (clients) from the providers of services by a well-defined encapsulating interface. The clients are isolated from the implementations of services as data representations and executable code. An object system provides services to the clients. A client of a service is an entity capable of requesting a service.

The concepts associated with the object semantics are:

Object An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

Requests Clients request services by issuing requests. A request is an event. The information associated with a request consists of an operation, a target object, zero or more parameters and an optional request context.

Value A value is anything that may be actual parameter in a request. A value may identify an object (object name) for the purpose of performing the request.

Object Reference An object reference is an object name that reliably denotes a particular object. It identifies the same object each time the reference is used in a request. An object may be denoted by multiple, distinct object references.

Object Creation and Destruction Objects can be created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

Type A type is an identifiable entity with an associated predicate defined over values. An object type is a type whose members are objects.

Interfaces An interface is a description of a set of possible operations that a client may request of an object. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface. Interfaces are specified in Interface Definition Language (IDL).

Operation An operation is an identifiable entity that denotes a service that can be requested. An operation has a signature that describes the actual values of request parameter and returned values.

Signature A signature consists of

- a specification of the parameters required in requests for that operation
- a specification of the result of the operation
- a specification of the exceptions that may be raised by a request for the operation and the types of the parameters accompanying them
- a specification of additional contextual information that may affect the request
- an indication of the execution semantics the client should expect from a request for the operation.

Parameters A parameter is characterized by its mode and its type. The mode indicates whether the value should be passed from the client to server (IN), from server to client (OUT), or both (IN OUT). The type constraints the possible value which may be passed in the direction dictated by the mode.

Return Result The return result is a distinguished OUT parameter.

Exceptions An exception is an indication that an operation request was not performed successfully.

Contexts A request context provides additional, operation-specific information that may affect the performance of a request.

A.2 Implementation Model

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. An object implementation provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. It typically includes definitions of the methods that operate upon the state of an object and information about the intended type of the object.

The implementation model consists the following two parts:

A.2.1 Execution Model

This model describes how the services are performed. A requested service is performed in a computational system by executing code that operates upon some data. The data represents a component of the state of the computational system. The code performs the requested service, which may change the state of the system. A method is the code that is executed to perform the service. When a client issues a request, a method of the target object is called. The input parameters are passed from the client to the method. The output parameters and the returned value (or an exception) are passed back to the client.

A.2.2 Construction Model

A computational object system must provide mechanisms for realizing the behavior of requests. These mechanisms include definitions of object state, definitions of methods and definitions of how the object infrastructure is to select the methods to execute and to select the relevant portions of object state to be made accessible to the methods.

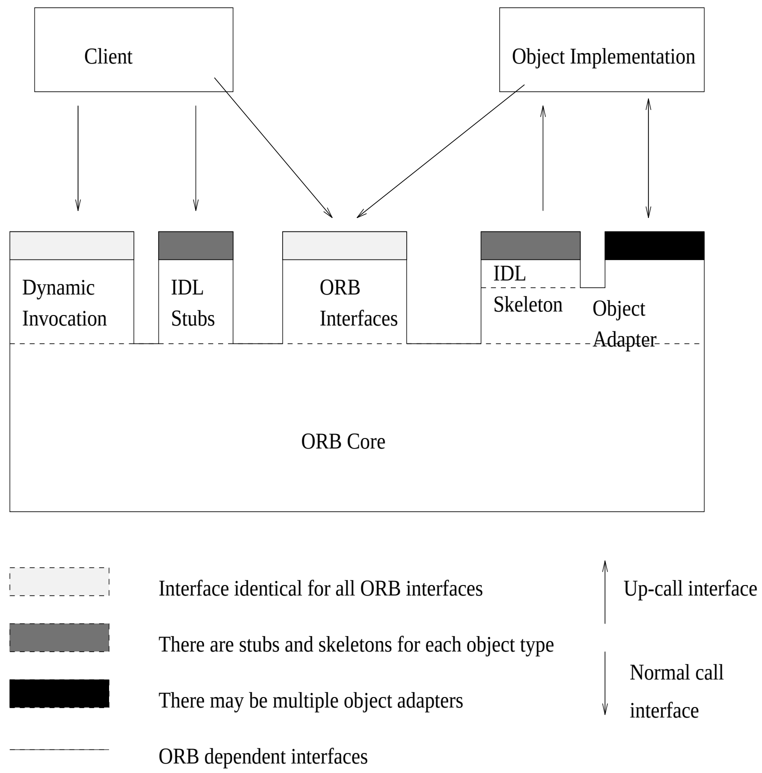

Figure A.1: **The Structure of Object Request Broker Interfaces**

A.3 The Structure of an Object Request Broker

The structure of an individual Object Request Broker (ORB) is shown in Figure A.1. The ORB is responsible for all mechanisms required to find the object implementation for request, to prepare the object implementation to receive the request and to communicate the data making up the request. The interface the client sees is completely independent of where the object is originally located, what programming language is implemented in or any other aspect which is not reflected in the object's interface.

To make a request, the Client can use the Dynamic Invocation Interface (DII) (independent of the target object) or an IDL stub (dependent on the interface of the target object) as shown in Figure A.1. The ORB locates the appropriate code, transmits the parameters and transfers the control to the Object Implementation. The Object Implementation receives a request as an up-call through the IDL skeleton. The implementation may also call the Object Adapter and the ORB while processing a request or at other times. The interface can be defined statically in IDL or dynamically. The interface is defined in IDL and/or in the Interface Repository. It is used to generate client stubs and the Object Implementation Skeletons. The Object Implementation information is provided at installation time and is stored in the Implementation Repository for the use during request delivery.

The main components of the Object Request Broker Architecture are:

Object Request Broker The ORB is not implemented as a single component but rather defined by its interfaces. The interface is organized in three categories,

- those operations that are the same for all ORB implementations.
- those operations that are specific to particular types of objects.
- those operations that are specific to particular styles of object implementations.

The ORB Core is the part of the ORB that provides the basic representation of objects and communication of requests. CORBA supports different object mechanisms by structuring the ORB with components above the ORB Core, which provide interface that can mask the differences between ORB Cores. In Figure A.2 different ORB implementations can be seen.

Client A client of an object has an access to an object reference for the object and invokes operations on the object. An object reference is a token that may be invoked or passed as a parameter to an invocation on a different object. Invocation of an object involves specifying the object to be invoked, the operation to be performed and the parameters to be given to the operation or returned from it.

Object Implementation An Object Implementation provides the actual state and behavior of an object. Besides defining the methods for the operations themselves, an implementation will usually define procedures for activating and deactivating objects and will use other objects or ORB facilities to make the object state persistent, to control access to the objects as well as implement the methods. The Object Implementation interacts with the ORB in a variety of ways to establish its identity, to create new objects and to obtain the ORB-dependent services. It primarily does this via access to an Object

Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation.

Basic Object Adapter An object adapter is the primary interface that an implementation uses to access ORB functions. The Basic Object Adapter (BOA) is an interface for generating object references, registering implementations that consist of one or more programs, activating implementations and authenticating requests. It also provides a limited amount of persistent storage for objects that can be used for connecting to a larger or more general storage facility, for storing access control information or other purposes. The BOA interface is expressed in IDL.

Interface Repository The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at runtime. The Interface Repository information may be used by the ORB to perform requests. Using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet be able to determine what operations are valid on the object and make an invocation on it. The Interface Repository is also a common place to store additional information associated with interfaces to ORB objects.

Implementation Repository The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. The Implementation Repository is the conventional place for recording such information. Installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

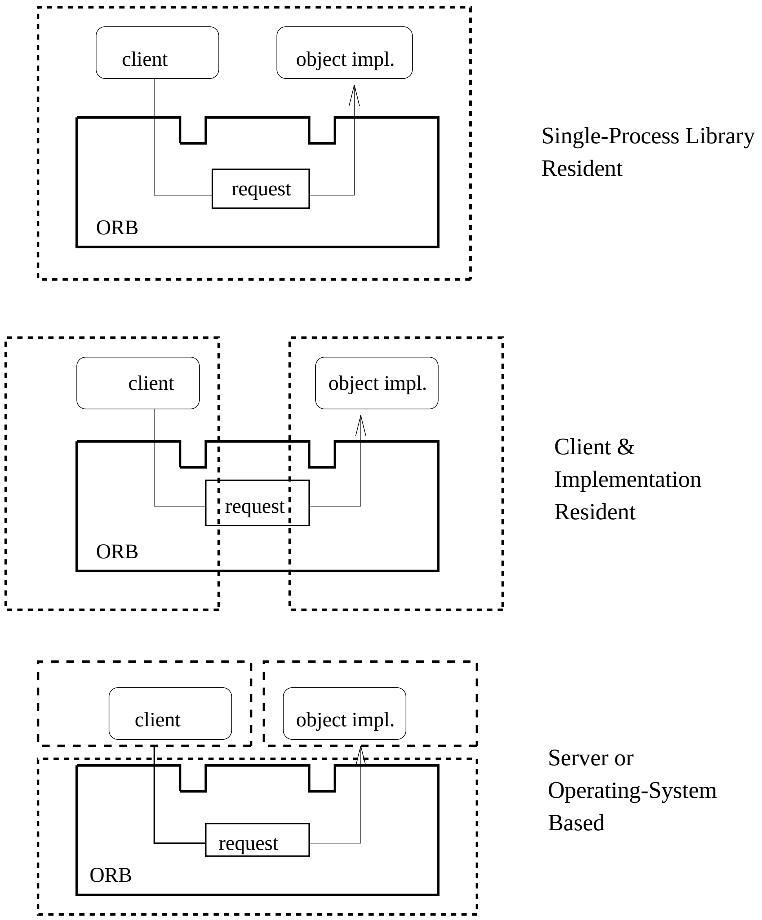

Figure A.2: **Different Object Request Broker Types**

A.4 Interface Definition Language (IDL)

The Interface Definition Language (IDL) is the language used to describe interfaces that the client objects call and object implementations provide. An interface consists of a set of named operations and the parameters to those operations. IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. This language is object-oriented and includes support for multiple inheritance. It is purely concerned with interface properties and does not provide any implementation information. From the IDL definitions, it is possible to map CORBA objects into particular programming languages, like C++, SmallTalk or object systems.

```
interface io {
        raw_data read(in long size) raises (access_denied,
                                  alerted, failure, end_of_data);
        void write(in raw_data data) raises (access_denied,
               alerted, failure, incomplete_write, end_of_data);
};
```

As it can be seen at the I/O example given above, IDL definitions are similar to C++ function prototypes. The interface `io` has two operations `read` and `write`, which can be invoked by clients. The operation `read` gets a parameter `size` of type `long` (which maps to `int` in C++) and returns a result of type `raw_data` and raises the standard exceptions `(access_denied, alerted, failure, end_of_data)` as defined in CORBA. The operation `write` gets a parameter `data` of type `raw_data` and does not return a result. Also, it raises the standard exceptions `(access_denied, alerted, failure, incomplete_write, end_of_data)` as defined in CORBA.

Appendix B

SPRING Operating System

The SPRING kernel provides an object-oriented process communication mechanism called *doors* [11], which are analogous to *ports* in Mach [1], [6]. A door is a communication endpoint, to which threads may execute cross address space calls. A domain that creates a door receives a door identifier, which it can pass to other domains, so that they can issue calls to the associated door. The SPRING kernel is responsible for all operations on doors and door identifiers, including their construction, destruction, copying and transmission.

Subcontract is the mechanism that SPRING utilizes to provide Inter-Object Communication. Most distributed systems present a model that objects reside at server machines and client machines possess object references that point to the object at the server. SPRING presents a model that clients are operating directly on objects, rather than object references. Instead of sending object reference of the server object, SPRING creates a copy of the server object and sends this copy to the client. However, both the objects point to the same underlying state and any changes on the objects are mapped to this underlying state. Subcontracts provide a communication buffer and the necessary operations to copy, destruct and transfer of the objects between the domains.

Stubs are the pre-defined and compiled language specific interfaces between the client and the subcontract buffer or the server and the subcontract buffer. The code for stubs is automatically generated from the IDL interface description given later in this section. When a client invokes a method of a pre-defined object, the client-side stub calls the server's subcontract (whose identity has been specified in the appropriate *environment file* and loads the arguments of the method into a buffer. The subcontract then invokes (the name of the method and the arguments to the method) the server-side stub: this stub unloads the arguments from the buffer, and invokes the server implementation code, where the method is executed. A result can be returned in the same way when the server processing has been completed. The diagram shown in Figure B.1 indicates the communication path between client and server applications in the RAACR.

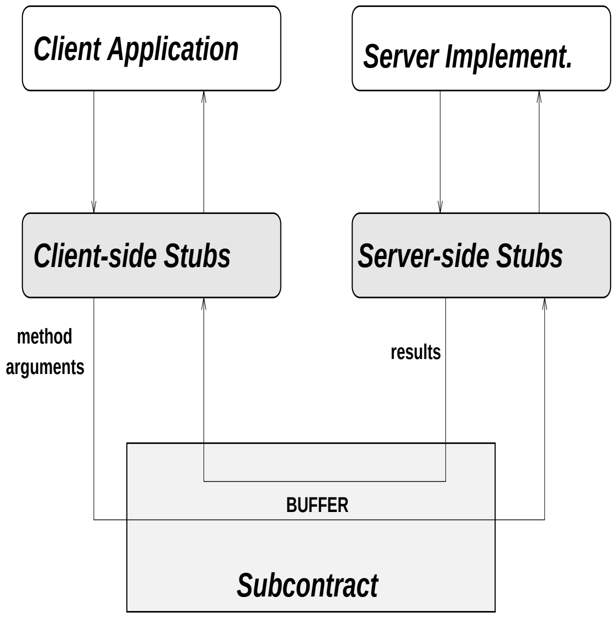

Figure B.1: **Spring Client-Server Transactions**

Appendix C

Data Dictionary for the RAACR

The member datas of the UIE:

- `char imagefile[20];`

  `imagefile` is the name of the input file containing the image.

- `FILE *fp_image;`

  `fp_image` is the pointer to the input image file.

- `int decision;`

  `decision` is the final detection result obtained from the Image Domain.

The member datas of the Images domain:

- `int xsize;`

  `xsize` is the width of the input image.

- `int ysize;`

  `ysize` is the height of the input image.

- `int image[xsize][ysize];`

  `Image` is a pointer to a 2-D matrix or image file. Each pixel may have a value between 0 and 255. However, for this application, each pixel may hold one of two possible values, 0 (black pixels) or 255 (white pixels), representing a binary image. The dimensions, width and height, of the `Image` is `xsize` and `ysize`, respectively.

- `float gradientmap[xsize][ysize];`

  `gradientmap` is a pointer to a 2-D matrix or data file (having same dimensions with the `image` matrix), which holds the direction of the intensity gradient vector, θ for the corresponding pixels in the `image` matrix.

- `int histogram[360];`

  `histogram` is a 1-D integer array of length `tetasize`. It holds the number of pixels for each value of possible gradient angles. Possible gradient angle values are between 0 and 359 and should be integer.

The member datas of the Histogram domain:

- `int tetasize;`

  `tetasize` shows the resolution of the histogram representing the $H(\theta)$ or in other words `tetasize` shows the sampling rate of the function $H(\theta)$. `tetasize` can vary between 10 and 360. The expected shape of the object in the image is also important in determining the `tetasize`. For an object with many edges, a small `tetasize` may be insufficient to recognize all the edges accurately.

- `typedef struct tetaclass {`

```
        int mean;
        int population;
        int variance;
} *classes;
```

`classes` is a pointer to a list of structure type `tetaclass`. Each element of this list represents a class in the θ histogram and consists three member variables: `mean` is the mean of the class, `population` is the number of pixels in the class and `variance` is the variance of the class. It is determined to be a list rather than an array since the number of edges of the object is not known beforehand.

- `int no_classes;`

  `no_classes` is the number of edge classes found by the Knowledge Source *FindEdges*. This is not the final edge number. The Knowledge Source *IdentifyObject* scans through the classes and determines the classes eligible to be an edge.

- `int Decision;`

  `decision` is Boolean, or integer as it is defined in C/C++. Holds 1 if the decision is "YES"(means, the object in the image is a square), holds 0 if the decision is "NO".

Appendix D

Source Code

D.1 image.idl

```
//           Interface Definitions for image domain domain
interface images {
void SetDimensions (copy long xsize, copy long ysize);
void SetPixel (copy long x, copy long y, copy long value);
long ActivateImage ();
 };
```

D.2 histogram.idl, histogram.env

```
//          Interface Definitions for histogram domain methods
interface histogram {
void SetRange (copy long range);
        void SetHistogram (copy long teta, copy long number);
long ActivateHistogram (copy long edge);
};
```

```
$singleton: sc=singleton sc_file="spring/subcontracts/singleton.h";
histogram: file = "histogram.idl";
histogram: $singleton;
```

D.3 uie.cc

```
//      uie.cc    Copyright (c) 1996 Yonet Eracar
//  Include statements ..
#include <spring/util/spring_lib.h>
#include <spring/util/stdio.h>
#include <spring/streams/stream.h>
#include <spring/naming/naming.h>
#include <spring/util/math.h>
#include "image.h"

int
main()
{
    // Resolve and get the object reference ..
    int i;
    images_fp m;

    for(i=0; i<10; i++) {
       try {
           m = naming<images>::resolve(domain_environment,
       "machine/services/images");
```

```
            break;
        }
        catch (naming_context_not_found) {
            (void)printf("Waiting for 'images' to be published\n");
        }
    }

    if (m == 0) {
        (void)printf("Could not look up "images"\n");
        exit(1);
    }

    int
        finish = 0,       // Token to finish, default =0
        x, y,             // Dimensions of the image ..
        xcenter, ycenter, // Center of the object ..
        delx, dely,       // Width, height ..
        obj_type,         // Type of the object
        j, k, count,
        temp,
        org_image[50][50];  // Input image
    float angle,            // Rotation angle
          distance;         // Distance from the center
    char filename[10];
    FILE *fp_inp;

    //  Enter the loop, if finish = 1, finish execution ..
```

```
while (!finish) {

   // Get the inputs from the user ...
   printf("\nuie:Enter the width (<50) of image file:");
   scanf("\n%d", &x);

   printf("\nuie:Enter the height (<50) of image file:");
   scanf("\n%d", &y);

   printf("\nuie:Enter the xcenter of object:");
   scanf("\n%d", &xcenter);

   printf("\nuie:Enter the ycenter of object:");
   scanf("\n%d", &ycenter);

   // Initialize the image matrix in Image domain..
   m->SetDimensions(x ,y);

   // Ask whether it is a rectangle or a circle ..
   printf("\nuie:Enter the number of object you want:");
   printf("\n     1)Square");
   printf("\n     2)Rectangle");
   printf("\n     3)Object file\n");
   scanf("\n%d", &obj_type);

   // Create the input image according to the input you get from
```

```
// the user ...

switch(obj_type) {
case 1: case 2:       // If it is a square or a rectangle

    // Get the height and width of the square or the rectangle
    printf("\nuie:Enter the delx of object:");
    scanf("\n%d", &delx);

    printf("\nuie:Enter the dely of object:");
    scanf("\n%d", &dely);

    // This loop sets the image matrix ....
    for(i=0; i<y; i++)
      for(j=0; j<x; j++)
         {
          if((i<ycenter-dely) || (i>ycenter+dely) ||
             (j>xcenter+delx) || (j<xcenter-delx))
                   org_image[i][j] = 255;
          else
                   org_image[i][j] = 0;
         }
    break;

 case 3:              // If it is a circle ...

// Get the radius ...
```

```
    printf("\nuie:Enter the name of the image file:");
    scanf("\n%s", &filename);

    // Initialize the data ..
    for (i=0; i<y; i++)
       for(j=0;j<x;j++)
          org_image[i][j] = 0.0;

    // Open and read the file
    if ((fp_inp = fopen(filename, "r")) == NULL)
    {
     printf("Error, in reading file.");
     exit(0);
    }
    fscanf(fp_inp,"%d\n", &count);
    for (k=0; k<count; k++){
       fscanf(fp_inp,"%d %d", &i, &j);
       org_image[i][j] = 255;
    }
    fclose(fp_inp);
    break;

 }

 // Load the Image Domain with the data ...
 for(i=0; i<y; i++)
   for(j=0; j<x; j++)
```

```
            m->SetPixel(j,i,org_image[i][j]);

       // Activate Image, Histogram and display the decision ...
       if(m->ActivateImage() == 1) {
            printf("\nuie:The object in the image is a SQUARE.\n");
       }
       else {
            printf("\nuie:The object in the image is NOT a SQUARE.\n");
       }

       printf("\nuie:Do you want to finish?(1 for YES)\n");
       scanf("\n%d", &finish);
   }

    // Destroy the object ..
    m->consume();
    return(0);
}
```

D.4 image_server.cc

```
//      image_server.cc    Copyright (c) 1996 Yonet Eracar
//  Include statements ..
#include <spring/util/spring_lib.h>
#include <spring/util/stdio.h>
#include <spring/streams/stream.h>
#include <spring/naming/naming.h>
```

```
#include "image_impl.h"

int main()
{
    images_impl *hfi = new images_impl ();
    images_fp h = hfi->make_fp ();

    try {
        naming<images>::rebind (domain_environment,
                                "machine/services/images",
                                h);
        printf("The bind of images has been completed.\n");
    }
    catch (...) {
        printf("The bind of images has failed.\n");
    }
    printf("before becoming server.\n");
    springlib::become_server ();
    printf("after  becoming server.\n");
}
```

D.5 histogram_server.cc

```
//      histogram_server.cc    Copyright (c) 1996 Yonet Eracar
//  Include statements ..
```

```
#include <spring/util/spring_lib.h>
#include <spring/util/stdio.h>
#include <spring/streams/stream.h>
#include <spring/naming/naming.h>
#include "histogram_impl.h"

int main()
{
    histogram_impl *hi = new histogram_impl ();
    histogram_fp h = hi->make_fp();

    try {
        naming<histogram>::rebind (domain_environment,
                                   "machine/services/histogram",
                                   h);
        printf("The bind of histogram has been completed.\n");
    }
    catch (...) {
        printf("The bind of histogram has failed.\n");
    }

    springlib::become_server ();
}
```

D.6 image_impl.h

```
// image_impl.h generated by contoimpl (3.11.2)
// from image.idl
// Sun Jun  2 16:36:54 1996

#ifndef image_impl_h
#define image_impl_h
#pragma interface

#include "image_srvr.h"

#define MAX 20
#define TETA 360

class images_impl:
    public images_srvr
{
public:
    virtual void SetDimensions(int arg_xsize, int arg_ysize) throw ();
    virtual void SetPixel(int arg_x, int arg_y, int arg_value) throw ();
    virtual int ActivateImage() throw ();

// Member data
private:
    int xsize, ysize,
        data[MAX][MAX],
```

```
        graddat[MAX][MAX],
        histdat[TETA];

// Member methods inaccessable by the UIE
private:
    void SobelEdgeFinder(int arg_dir1);
    void LaplacianEdgeFinder();
    void CreateHistogram();
};
#endif not image_impl_h
```

D.7 image_impl.cc

```
// image_impl.cc generated by contoimpl (3.11.2)
// from image.idl
// Tue May 14 23:24:24 1996

// Includes
#include <spring/util/spring_lib.h>
#include <spring/util/string.h>
#include <spring/naming/naming.h>
#include <spring/util/stdio.h>
#include <spring/streams/stream.h>
#include "histogram_clientimpl.h"
#include "image_impl.h"
```

```
// Macros
#define yabs(x) ((x) < 0 ? -(x) : (x))

void images_impl::SetDimensions(int arg_xsize, int arg_ysize) throw ()
    // interface ::images
    // operation void SetDimensions(copy long xsize, copy long ysize)
{
    int i;

    printf("\nimage:Now, in SetDimensions..");

    // Initialize the dimensions ..
    xsize = arg_xsize;
    ysize = arg_ysize;

    printf("\nimage:Now, image domain is initialized.");
}

void images_impl::SetPixel(int arg_x, int arg_y, int arg_value) throw ()
    // interface ::images
    // operation void SetPixel(copy long x, copy long y, copy long value)
{
    data[arg_y][arg_x] = arg_value;
    // printf("\nimage:Now, in SetPixel.");
}

int images_impl::ActivateImage() throw ()
```

```
    // interface ::images
    // operation long ActivateImage()
{
    histogram_fp his = get_histogram_fp();

    int i, j,
        decision,
        detector,
        detection = -1,
        range = TETA,
        repetition = 0,   // Shows the # of repetition of execution
        sobel_direction,
        Sobel = 1,
        Laplacian = 2,
        temp;
    char filename[20] = "activateimage.dbs";

    FILE *fp_inp;

    // Read the State file and initialize the parameters ..

    // Open, read and close the file
    if ((fp_inp = fopen(filename, "r")) == NULL)
     {
        printf("Error, in reading file.");
        exit(0);
     }
```

```
fscanf(fp_inp,"%d %d", &detector, &sobel_direction);
fclose(fp_inp);

while (detection == -1 && repetition < 2) {

    printf("\nimage:Now, in ActivateImage ...");

    // Check whether it is the second round ..
    if (repetition > 0) {
        if (detector == Sobel)
            detector = Laplacian;
    else if (detector == Laplacian)
            detector = Sobel;
    }
    // Increase repetition count ..
    repetition++;

    if (detector == Sobel)
        SobelEdgeFinder(sobel_direction);
    else
        LaplacianEdgeFinder();

    CreateHistogram();

    // Initialize the Histogram domain
    his->SetRange(range);
```

```
        // Copy the Histogram ...
        printf("\nimage:Now, ActivateImage invokes SetHistogram...");

        // Pass only the hist values other than 0 ....
        for (i=0; i<range; i++) {
           if (histdat[i] != 0)
              his->SetHistogram(i,histdat[i]);
        }

        // Identify the object ...
        detection = his->ActivateHistogram(4);

        // Make a decision ..
        if (detection == 4)
            decision = 1;
        else
            decision = 0;
    }

    // Open, write and close the file
    if ((fp_inp = fopen(filename, "w")) == NULL)
{
    printf("Error, in writing file.");
    exit(0);
}
    fprintf(fp_inp,"%d %d", detector, sobel_direction);
    fclose(fp_inp);
```

```
    return decision;
}

void images_impl::SobelEdgeFinder(int arg_dir1)
    // interface ::images
    // operation void SobelEdgeFinder(copy char dir1)
{

int i, j,            // Indices for the loops
    type;

float tempteta,
      gy, gx, upper = 0.0,
      R_TO_D = 57.2957,
      threshold,
      arctan,
      x,               // slope (gy/gx)
      v[MAX][MAX];     //Temporary image array..

printf("\nimage:Now, at Sobel edge detector.");

 for(i=0; i<ysize; i++)
   for(j=0; j<xsize; j++)
    {
     if((i==0) || (j==0) || (i==ysize-1) || (j==xsize-1))
       {
  tempteta = -1.0;
```

```
      }
    else
      {
  if(arg_dir1 == 1) {
      gy = data[i+1][j-1] + 2*data[i+1][j]
                 + data[i+1][j+1] - data[i-1][j-1]
         - 2*data[i-1][j] - data[i-1][j+1];
      gx = data[i+1][j+1] + 2*data[i][j+1]
         + data[i-1][j+1] - data[i+1][j-1]
         - 2*data[i][j-1] - data[i-1][j-1];
  }
  else if(arg_dir1 == 2) {
      gy = - 2*data[i+1][j-1] - data[i+1][j]
                 - data[i][j-1] + data[i-1][j]
         + 2*data[i-1][j+1] + data[i][j+1];
      gx = 2*data[i+1][j+1] + data[i][j+1]
         + data[i+1][j] - data[i][j-1]
         - 2*data[i-1][j-1] - data[i-1][j];
  }

  v[i][j] = yabs(gx) + yabs(gy);
  if(v[i][j] > upper) upper = v[i][j];

  // Find the gradient angle image ....

  if(gx == 0.0 && gy == 0.0) tempteta = -1 ;
  else
```

```
      {
                if(gy>=0 && gx>gy) type = 1;
                else if(gx>=0 && gy>gx) type = 2;
                else if(gx>=0 && -gy>gx) type = 3;
                else if(gx>0 && gy<0 && -gy<gx) type = 4;
                else if(gx<0 && gy<gx) type = 5;
                else if(gy>=0 && -gx>gy) type = 6;
                else if(gy<0 &&  gx<gy) type = 7;
                else if(gy>0 &&  gx<0 && gy>-gx) type = 8;

                if(type==1 || type==4 || type==6 || type==7)
                      x = gy/gx;
if(type==2 || type==3 || type==5 || type==8)
                      x = gx/gy;

                arctan = x
                       - x*x*x/3.0 + x*x*x*x*x/5.0
                       - x*x*x*x*x*x*x/7.0
                       + x*x*x*x*x*x*x*x*x/9.0
                       - x*x*x*x*x*x*x*x*x*x*x/11.0;

                switch(type) {
                    case 1: tempteta = R_TO_D*arctan;
                            break;
                    case 2: case 8:
                            tempteta =  90 - R_TO_D*arctan;
                            break;
```

```
                    case 4: tempteta =  360 + R_TO_D*arctan;
                            break;
                    case 5: case 3: tempteta =  270 - R_TO_D*arctan;
                            break;
                    case 6: case 7: tempteta =  180 + R_TO_D*arctan;
                            break;
                }
           }

}
     graddat[i][j] = (int)tempteta;
     }

printf("\nEnd of edge detector.");
threshold = upper/2.0;

for(i=0; i<ysize; i++)
   for(j=0; j<xsize; j++)
       {
         if (v[i][j] < threshold)
          graddat[i][j] = -1;
       }
}


void images_impl::LaplacianEdgeFinder()
    // interface ::images
```

```
    // operation void LaplacianEdgeFinder()
{

int i, j;
int type;
float tempteta;
float gy, gx, upper = 0.0;
float R_TO_D = 57.2957;
float threshold;
float arctan;
float x;              // slope (gy/gx)

float v[150][150];    //Temporary image array..

for(i=0; i<ysize; i++) {
   for(j=0; j<xsize; j++)
    {
     if((i==0) || (j==0) || (i==ysize-1) || (j==xsize-1))
       {
         tempteta = -1.00;
       }
     else
       {
         gy = - 2*data[i+1][j-1] - data[i+1][j]
              - data[i][j-1] + data[i-1][j]
              + 2*data[i-1][j+1] + data[i][j+1];
         gx = 2*data[i+1][j+1] + data[i][j+1]
```

```
          + data[i+1][j] - data[i][j-1]
          - 2*data[i-1][j-1] - data[i-1][j];

    v[i][j] = yabs(gx) + yabs(gy);
    if(v[i][j] > upper) upper = v[i][j];

    // Find the gradient angle image ....
    if(gx == 0.0 && gy == 0.0) tempteta = -1 ;
    else
        {
          if(gy>=0 && gx>gy) type = 1;
          else if(gx>=0 && gy>gx) type = 2;
          else if(gx>=0 && -gy>gx) type = 3;
          else if(gx>0 && gy<0 && -gy<gx) type = 4;
          else if(gx<0 && gy<gx) type = 5;
          else if(gy>=0 && -gx>gy) type = 6;
          else if(gy<0 &&  gx<gy) type = 7;
          else if(gy>0 &&  gx<0 && gy>-gx) type = 8;

          if(type==1 || type==4 || type==6 || type==7)
              x = gy/gx;
          if(type==2 || type==3 || type==5 || type==8)
              x = gx/gy;

          arctan = x
                  - x*x*x/3.0
                  + x*x*x*x*x/5.0
```

```
                    - x*x*x*x*x*x*x/7.0
                    + x*x*x*x*x*x*x*x*x/9.0
                    - x*x*x*x*x*x*x*x*x*x*x/11.0;

                switch(type) {
                     case 1: tempteta = R_TO_D*arctan;
                             break;
                     case 2: case 8: tempteta =  90 - R_TO_D*arctan;
                             break;
                     case 4: tempteta =  360 + R_TO_D*arctan;
                             break;
                     case 5: case 3: tempteta =  270 - R_TO_D*arctan;
                             break;
                     case 6: case 7: tempteta =  180 + R_TO_D*arctan;
                             break;
                }
            }

        }
     graddat[i][j] = (int)tempteta;
     }
  }

printf("\nupper = %f",upper);
threshold = upper/2.0;

for(i=0; i<ysize; i++)
```

```
   for(j=0; j<xsize; j++)
     {
       if (v[i][j] < threshold)
                graddat[i][j] = -1;
     }
}
```

D.8 histogram_impl.h

```
// histogram_impl.h generated by contoimpl (3.11.2)
// from histogram.idl
// Sun Jun  2 16:39:14 1996

#ifndef histogram_impl_h
#define histogram_impl_h
#pragma interface

#include "histogram_srvr.h"

// Structures

struct Classes {
    int mean,
        population,
        member[50],
```

```
        nomembers;
};


class histogram_impl:
    public histogram_srvr
{
public:
    virtual void SetRange(int arg_range) throw ();
    virtual void SetHistogram(int arg_teta, int arg_number) throw ();
    virtual int ActivateHistogram(int arg_edge) throw ();
private:
    int tetasize,
        hist[360],
        noclasses,
        decision;
    struct Classes tetaclass[20];
private:
    void FindEdges();
    int IdentifyObject(int arg_edge);
};


#endif not histogram_impl_h
```

D.9 histogram_impl.cc

```
// histogram_impl.cc generated by contoimpl (3.11.2)
```

```
// from histogram.idl
// Tue May 14 23:20:45 1996

// Include files

#include <spring/util/spring_lib.h>
#include <spring/util/string.h>
#include <spring/util/stdio.h>
#include <spring/naming/naming.h>
#include <spring/streams/stream.h>
#include "histogram_impl.h"

// Macros
#define MAX 360
#define yabs(x) ((x) < 0 ? -(x) : (x))

void histogram_impl::SetRange(int arg_range) throw ()
    // interface ::histogram
    // operation void SetRange(copy long range)
{
    int i;

    // Set the teta range ....
    tetasize = arg_range;

    // Set everything to 0 ..
    for (i=0; i<tetasize; i++)
```

```
        hist[i] = 0;
}

void histogram_impl::SetHistogram(int arg_teta, int arg_number) throw ()
    // interface ::histogram
    // operation void SetHistogram(copy long teta, copy long number)
{
    if(arg_teta<360 && 0<=arg_teta)
       {
          hist[arg_teta] = arg_number;
       }
}

int histogram_impl::ActivateHistogram(int arg_edge) throw ()
    // interface ::histogram
    // operation long ActivateHistogram(copy long edge)
{
    int edgenumber;

    printf("\nhistogram:Now in Activatehistogram..");
    FindEdges();
    edgenumber = IdentifyObject(arg_edge);

    return edgenumber;
}

int histogram_impl::FindPeaks()
```

```
    // interface ::histogram
    // operation void FindPeaks()
{
   int i, k, index,
       HistCopy[360],
       delta,
       max_peakcount = 10,
       peak,
       peakcount = 0,
       token;

    //++
    //  This parameter is related to the expected edge number
    //--
    delta = 30;

   //  Copy the histogram ..count the # of edge points
    for(i=0; i<MAX; i++)
        HistCopy[i] = hist[i];

    //  Find the pTeta ..initialize exemplars
    do {
        token = 0;
        peak = 0;    //  initialize peak count everytime
        for(i=0; i<MAX; i++)
            if(HistCopy[i] > peak) {
                peak = HistCopy[i];
```

```
                pTeta[peakcount].mean = i;
                token = 1;
            }
        if(token == 1) {
            pTeta[peakcount].population =
                hist[pTeta[peakcount].mean];
            pTeta[peakcount].member[0] =
                pTeta[peakcount].mean;
            pTeta[peakcount].nomembers  = 1;

            //   Put 0 to neighbors of the pTeta we find..
            for(k=-delta; k<=delta; k++) {
                index = pTeta[peakcount].mean + k;
                if(index < 0)
                    index = MAX + index;
                if(index >= MAX)
                    index = index - MAX;
                HistCopy[index] = 0;
            }
            //  Increase the peakcount
            peakcount++;
        }
    }
    while (peakcount<max_peakcount && token == 1);

    return peakcount;
}
```

```
void histogram_impl::FindMeans(int peakcount)
    // interface ::histogram
    // operation void FindMeans()
{
    int i, j, k, index, ind,
        distance,
        min_distance,
        newmember,
        HistCopy[MAX];

    // ReCopy HistCopy again
    for(i=0; i<MAX; i++)
        HistCopy[i] = hist[i];

    // Exclude the exemplars..
    for(k=0; k<peakcount; k++) {
        HistCopy[pTeta[k].mean] = 0;
        printf("\n%d, mean: %d, pop: %d", k,
                 pTeta[k].mean, pTeta[k].population);
    }
    // Scan all teta values in histogram and classify ..
    for(i=0; i<MAX; i++) {
        min_distance = MAX;
        if(HistCopy[i] != 0) {

            //  Check with each cluster ...
```

```
    for(j=0; j<peakcount; j++) {
        distance = yabs((pTeta[j].mean)-i);
        if(distance > MAX / 2)
             distance = MAX  - distance;

        if(min_distance > distance) {
            min_distance = distance;
            ind = j;
        }
    }
    // Check the negativity ...
    distance = yabs((pTeta[ind].mean)-i);
    if (distance > MAX / 2) {
        if (pTeta[ind].mean > i)
            newmember = MAX + i;
        else
            newmember = i - MAX;
    }
    else
         newmember = i;

    //  Modify the mean of the cluster ...
    pTeta[ind].mean = (pTeta[ind].mean
        * pTeta[ind].population +  HistCopy[i]*newmember)
        / (pTeta[ind].population + HistCopy[i]);
    pTeta[ind].population +=  hist[i];
    pTeta[ind].member[pTeta[ind].nomembers] = newmember;
```

```
            pTeta[ind].nomembers  += 1;
        }  // if
    }  // for(i=0; ....

    // Display the means ...
    for(k=0; k<peakcount; k++)
        printf("\n%d, mean: %d", k, pTeta[k].mean);
}

void histogram_impl::FindVariances(int peakcount)
    // interface ::histogram
    // operation void FindVariances()
{
    int i, j, index;

    for(j=0; j<peakcount; j++) {
        pTeta[j].variance = 0;
        for(i=0; i<pTeta[j].nomembers; i++) {
            if (pTeta[j].member[i] < 0)
                index = MAX + pTeta[j].member[i];
            else if (pTeta[j].member[i] >= MAX)
                index = pTeta[j].member[i] - MAX;
            else
                index = pTeta[j].member[i];

            pTeta[j].variance += hist[index] *
               (pTeta[j].mean - pTeta[j].member[i])
```

```
               * (pTeta[j].mean - pTeta[j].member[i]);
        }
    }
}

void histogram_impl::SecondScan(int peakcount)
    // interface ::histogram
    // operation void SecondScan()
{
    int i, j, index,
         total;

    for(j=0; j<peakcount; j++) {
        total = 0;
        for(i=0; i<pTeta[j].nomembers; i++) {
            if (pTeta[j].member[i] < 0)
                index = MAX + pTeta[j].member[i];
            else if (pTeta[j].member[i] >= MAX)
                index = pTeta[j].member[i] - MAX;
            else
                index = pTeta[j].member[i];
            total += hist[index] * pTeta[j].member[i];
            }
        pTeta[j].mean = total / pTeta[j].population;
     }
}
```

```
void histogram_impl::FindEdges()
    // interface ::histogram
    // operation void FindEdges()
{
    int j,  peakcount = 0;

    //   pTeta[i].mean  = mean  of the class
    //   pTeta[i].population  = population  of the class
    //   pTeta[i].member[j]  = members  of the class
    printf("\nhistogram:Now, in FindEdges..");

    // Find the peaks ..
    peakcount = FindPeaks();
    printf("\npeakcount =%d",peakcount);

    // Find the class members and the means
    FindMeans(peakcount);

    // Perform a second Scan and find the actual means ..
    SecondScan(peakcount);

    // Find the variances...
    FindVariances(peakcount);

    noclasses = peakcount;
}
```

```
int histogram_impl::SortPopulation(float thrUp, float thrLow,
                                   int size)
    //  sorts the edge classes according to
    //  their population
{
    int i, j,
        edges = 0,
        total = 0;
    float totprob = 0,
          marprob = 0.25;

    // Sort the classes in descending order first ..
    for(i=0; i<size-1; i++)
        for(j=0; j<size-1-i; j++)
              if (pTeta[j].population < pTeta[j+1].population) {
                   SwapClass(j,j+1);
               }

    // find the total number of edge points
    for(i=0; i<size; i++) {
        total += pTeta[i].population;
    }

    // Find the marginal probabilities of each additional edge
    //  and check if they qualify to be an edge ..
    while (marprob < thrUp && marprob > thrLow && edges < size) {
```

```
        marprob = pTeta[edges].population/(1.0*total);
        if(marprob > thrLow)
              edges++;
    };

    return edges;
}


void histogram_impl::SortMean(int size)
    //  sorts the edge classes according to
    //  their mean
{
       int i, j;

       // sort according to the angles
       for(i=0; i<size-1; i++) {
           for(j=0; j<size-1-i; j++)
                if (pTeta[j].mean > pTeta[j+1].mean) {
                    SwapClass(j, j+1);
                }
        }
}


float histogram_impl::Sqroot(float base, float error)
{
```

```
    float dividend,
           err = 1000,
           est;

     if (base > 1.0)
          est = base / 2.0;     // best estimate if base  > 1
     else
          est = (1 + base) / 2.0; // b/w 1 and base ..

     while (err > error) {
         dividend = base / est;
         err = yabs (est - dividend);  // yabs() is implemented before
         est = (est + dividend) / 2.0;
     }

     return (est);
}

void histogram_impl::SwapClass(int i, int j)
{
    struct Classes temp;

    // swap means
    temp.mean = pTeta[i].mean;
    pTeta[i].mean = pTeta[j].mean;
    pTeta[j].mean = temp.mean;
```

```
    // swap populations
    temp.population = pTeta[i].population;
    pTeta[i].population = pTeta[j].population;
    pTeta[j].population = temp.population;

    // swap variances
    temp.variance = pTeta[i].variance;
    pTeta[i].variance = pTeta[j].variance;
    pTeta[j].variance = temp.variance;
}


int histogram_impl::DecisionRule(int *D, int *UL, int *LL, float *sigD)
    //
    //   Decision rule is applied for all edges ....
{
    int i,
        T;

    //  Apply Decision Rule to all edges
    //  using T distribution

    for (i=0; i<4; i++) {
        T = (int)((D[i] - 90) / sigD[i]);  // calculate t statistic
        if(T > UL[i] || T < LL[i])
            return 0;             // all angles arent right
    }
```

```
    return (1);
}

void histogram_impl::FindDifferences(int *D)
{
    int i;

    // Find the D_i ,  angles b/w edges ..
    for (i=0; i<4; i++) {
        if (i == 0)
            D[i] = MAX + pTeta[i].mean - pTeta[3].mean;
        else
            D[i] = pTeta[i].mean - pTeta[i-1].mean;
        printf("\nhist:   D[%d]= %d",i,D[i]);
    }
}

void histogram_impl::FindSigmaD(float *SigmaD)
       // Smith-Satterthwaite method, pooling
       //
{
    int i;
    float error = 0.00005,
          pool;

    // Find the D_i ,  angles b/w edges ..
    for (i=0; i<4; i++) {
```

```
if (i == 0)
    pool = pTeta[i].variance/pTeta[i].population
 + pTeta[3].variance/pTeta[3].population;
else
    pool = pTeta[i].variance/pTeta[i].population
                 + pTeta[i-1].variance/pTeta[i-1].population;
SigmaD[i] = Sqroot(pool, error);
printf("\nhist:   SigD[%d]= %f",i,SigmaD[i]);
    }
}

void histogram_impl::FindLimits(int *UL, int *LL, float *alpha)
{
    int i;
    float s1, s2,
   n1, n2,
           t, x, y, dof;

    // Update the UL and LL for hypothesis test ..
    for (i=0; i<4; i++) {
        // Find the degrees of freedom
        s1 = pTeta[i].variance;
        n1 = (float)(pTeta[i].population);
        if(i ==0) {
            s2 = pTeta[3].variance;
            n2 = (float)(pTeta[3].population);
        }
```

```
        else {
            s2 = pTeta[i-1].variance;
            n2 = (float)(pTeta[i-1].population);
        }

        dof = (s1*s1/n1 + s2*s2/n2) * (s1*s1/n1 + s2*s2/n2)
               /  ( (s1*s1/n1)*(s1*s1/n1)/(n1-1) +
               (s2*s2/n2)*(s2*s2/n2)/(n2-1) );

        // Ask T  according to alpha value and dof ...
        printf("\nalpha=%f, dof=%f, find T.\n",
              alpha[i]/2.0, dof);
        scanf("\n%f",&t);
        UL[i] = (int)t;
        LL[i] = (int)(0 - t);
    }
}

void histogram_impl::UpdateParameters(int sortresult, int result,
 float prob, int step,float probMarU, float probMarL, float *alpha)
{
    int feedback;
    float correct, newPCR, newMarL, newMarU;
    char filename[20] = "identify.dbs";   // State info file
    FILE *fp_inp;

    // Write the result ..
```

```
if(result == 4) {
    printf("\nDecision: It is a SQUARE. Enter feedback.");
    decision = 1;
}
else {
    printf("\nDecision: It is NOT. Enter feedback.");
    decision = 0;
}
printf("Enter 1 for success, 0 for failure.\n");
// Get the feedback ..
scanf("\n%d", &feedback);

// Modify the PCR
if(step == 1)
    newPCR = feedback;
else {
    correct = prob*(step-1) + feedback;
    newPCR = correct / (float)step;
    printf("\nPCR(new) = %f", newPCR);
}
// Modify frame number ...
step++;

// Modify probMar ...
if (feedback == 1)   {   // if it is success
    newMarL = probMarL;
   newMarU = probMarU;
```

```
    }
    else  {                  // if it is a failure
        if (sortresult < 4){
             newMarL = probMarL - 0.02;
            newMarU = probMarU + 0.02;
        }
        else {
            newMarL = probMarL + 0.02;
           newMarU = probMarU - 0.02;
        }
    }

    // Open, write the new parameters and close the file
    if ((fp_inp = fopen(filename, "w")) == NULL)  {
             printf("Error, in reading file.");
             exit(0);
    }
    fprintf(fp_inp,"%f %d %f %f", newPCR, step, newMarU, newMarL);
    fprintf(fp_inp,"\n%f %f %f %f", alpha[0],alpha[1],alpha[2],alpha[3]);
    fprintf(fp_inp,"\n%d %d",feedback, decision);
    fclose(fp_inp);
}

float histogram_impl::FindGain(float PCR, float *alpha, float sigD)
    // interface ::histogram
    // operation float FindGain()
{
```

```
    float K;
    K = 0.02 * sigD;
    return K;
}

void histogram_impl::FindAlphas(float K, int sign, int dec,
float *alpha)
    // interface ::histogram
    // operation long FindGain()
{
    int i,
        omega;
    // Calculate omega function based on decision and feedback
    if (sign) omega = 0;    // No change, we are doing fine
    else {
        if (dec) omega = 1;   // Increase the limits
        else  omega = -1;          // Decrease the limits
    }
    // Re calculate the alpha values ...
    for(i=0; i<4; i++) {
        alpha[i] = K*omega + alpha[i];
    }
}

int histogram_impl::IdentifyObject(int arg_edge)
    // interface ::histogram
    // operation long IdentifyObject()
```

```
{
    int i,                          // Indices
        edgenumber,                    // Final edge number
        rightangles,
        sign,
        sortresult,
        D[4],             // angle difference
        LL[4], UL[4],      // Confidence Interval ...
        rep;               // # of images so far

    char filename[20] = "identify.dbs";   // State info file

    float alpha[4],               // confidence constant
        K,                             // Gain constant
        sigD[4],           // variance of D samples
        confidence,
        probMarU, probMarL,
        PCR;              // Prob. Corr. reg so far
    FILE *fp_inp;

    printf("\nhistogram:Now in IdentifyObject");

    // Read the State file and initialize the parameters ..
    if ((fp_inp = fopen(filename, "r")) == NULL) {
          printf("Error, in reading file.");
          exit(0);
    }
```

```
fscanf(fp_inp,"%f %d %f %f\n", &PCR, &rep, &probMarU, &probMarL);
fscanf(fp_inp,"%f %f %f %f", &alpha[0], &alpha[1],
        &alpha[2],&alpha[3]);
fscanf(fp_inp,"%d %d", &sign, &decision);
fclose(fp_inp);

// Sort the edges and find the number of edges
sortresult = SortPopulation(probMarU, probMarL, noclasses);
printf("\nSort: result = %d\n", sortresult);

// Return to Image Domain without changing anything ...
if(sortresult > 5 || sortresult < 3)
    return (-1);

if (sortresult == 4) {
    // sort the edges according to their angles ..
    SortMean(sortresult);

    // Find the angles ..
    FindDifferences(D);

    // Find the sigma(D)'s..
    FindSigmaD(sigD);

    // Controller Designer ....
    K = FindGain(PCR, alpha);
```

```
        // Find the alpha values according to PCR
        // Control Law ...
        FindAlphas(K, sign, decision, alpha);

        // Find ULs and LLs given alpha and sigmas
        FindLimits(UL, LL, alpha);

        // Apply the Decision Rule
        rightangles = DecisionRule(D, UL, LL, sigD);

        if (rightangles == 1 )
             edgenumber = 4;           //  OK it is a Square
        else edgenumber = 5;    // No, it is not a square ..
    }

    // Get feedback, Update the PCR and rep
    UpdateParameters(sortresult, edgenumber, PCR, rep,
     probMarU, probMarL, alpha);

    return edgenumber;
}
```

Bibliography

[1] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proc. of the 1986 Summer Usenix Conference*, Atlanta, June 1986.

[2] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, 1986.

[3] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley Publishing Company, Reading, MA., 1991.

[4] R.J. Costello. Metrics for requirements engineering. Master's thesis, California State University, Long Beach, 1994.

[5] A. M. Davis. *Software Requirements: Objects, Functions, and States.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[6] R. Draves. A revised rpc interface. In *Proc. of the Usenix Mach Workshop*, Burlington, VT, October 1990.

[7] Y. E. Eracar, C. DiMarzio, and M. M. Kokar. False alarm reduction of ground penetrating radar returns through multisensor fusion. Technical report, Center for Electromagnetics Research, Northeastern University, MA, May 1996.

[8] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. The hearsay-ii speech-understatnding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, 1980.

[9] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[10] R. Gu. Formal specification and verification of sensor data fusion systems. Master's thesis, Northeastern University, Boston, MA, 1996.

[11] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. In *Proc. of the 1993 Summer Usenix Conference*, Cincinnati, June 1993.

[12] G. Hamilton, M.L. Powell, and J.K. Mitchell. Subcontract: A flexible base for distributed programming. Technical report, Sun Microsystems Laboratories, Inc., Mountain View, CA, 1995.

[13] R. Hayes-Roth. Architecture-based acquisition and development of software: Guidelines and recommendations from the arpa dssa program. Technical report, Teknowledge Federal Systems, Palo Alto, CA 94303, 1994.

[14] R. Hayes-Roth and W. Tracz. Dssa tool requirements for key process functions, version 2.0. Technical report, Teknowledge Federal Systems, Palo Alto, CA 94303, 1994.

[15] B. Isenstein. An eye to the future. *Defense Electronics*, pages 23–24, February 1995.

[16] M. Kokar. Learning control: Methods, needs and architectures. In K. M. Passino and P.J. Antsaklis, editors, *Introduction and Autonomous Control*, pages 263–282. Kluwer Academic Publishers, 1993.

[17] M. Kokar and K. Kim. Review of multisensor data fusion architectures and techniques. *Symposium on Intelligent Control*, pages 261–266, August 1993.

[18] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium-1*, pages 281–297, 1967.

[19] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, revision 1.1 edition, December 1993.

[20] Institute of Electrical and Electronics Engineers. Ieee standard glossary of software engineering terminology, ieee standard 982-1988. Piscataway, NJ, 1988.

[21] Institute of Electrical and Electronics Engineers. Ieee standard glossary of software engineering terminology, ieee standard 610.12. Piscataway, NJ, 1990.

[22] R. A. Paul. Metrics to improve the us army software development process. In R. Thayer and M. Dorfman, editors, *The First International Software Metrics Symposium*, pages 40–60, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[23] A. Rosenfeld. *Divide-and -Conquer Techniques for Detecting and Describing Global Features in Images*. Image Understanding. Ablex Publishing, New Jersey, 1989.

[24] A. Rosenfeld and J. M. Jolion. *A pyramid framework for early vision: multiresolutional computer vision.* Kluwer Academic Publishers, Boston, MA, 1994.

[25] E. Rumbaugh. Controlling propagation of operations using attributes on relations. In *Proc. of OOPSLA '88*. ACM, 1988.

[26] J.D. Sailor. System and software requirements engineering. In R. Thayer and M. Dorfman, editors, *System engineering: an introduction*, pages 35–47. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[27] Sun Microsystems, Inc., Mountain View, CA. *Spring Research Distribution 1.1 Technical Manual*, 1995.

[28] W. Tracz. Collected overview reports from the dssa project. Technical report, Loral Federal Systems, Owego, NY 13827, 1995.