An Architecture for Software that Adapts to Changes in Requirements

Yönet A. Eracar and Mieczyslaw M. Kokar
yeracar@coe.neu.edu and kokar@coe.neu.edu
Northeastern University
360 Huntington Avenue, Boston, MA 02115

August 31, 1998

Abstract

The goal of the research presented in this paper was to study a new software paradigm – *adaptive software* – in which the structure of an adaptive program is patterned upon the structure of an adaptive controller. Towards this aim, we implemented a domain-specific (object/target recognition) program (RAACR) that can adapt to changes in software requirements through the incorporation of feedback. RAACR is a hierarchy of domains (blackboards). Each domain includes multiple knowledge sources (KSs) and a domain scheduler. In response to feedback, knowledge sources change their processing parameters, while domain schedulers change the scheduling policy of the knowledge sources. A generic communication mechanism is implemented on the CORBA compliant SPRING operating system. The adaptability of the program is evaluated quantitatively using a requirements volatility measure and the probability of correct recognition.

1 Introduction

In object/target recognition applications, it is difficult to predict all possible scenarios that the sensors would have to deal with. In addition, the signal processing hardware, due to the complexity of processing, introduces 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 software fails to achieve its goal, i.e., recognize objects/targets when unexpected changes in the inputs occur. To deal with this kind of a problem, without rebuilding the software, the software developer would have to add some additional functionality to the software so that it would be able to adapt to the changes during its operation (run-time adaptability). The question is what kind of functionality would achieve the desired result?

Control engineers are faced with a similar problem: they have to design controllers that deal with input disturbances that are difficult to predict at the design time of the controlled system (in control terminology called *plant*). The main idea of control is to utilize *feedback* (a measure of performance of the system) and to add some *redundancy* to the system (this redundancy is called *controller*). Depending on the control paradigm, feedback is used to determine control inputs (*feedback control*), adjust the parameters of the *model* of the controlled plant and of the *control law* (*adaptive control* (Åström, 1989)), select or reconfigure a control law (*gain scheduling* (Shamma, 1996) and *reconfigurable control* (Montoya et al., 1982)). The area of control that deals with unexpected disturbances is called *intelligent control* (cf. (Antsaklis, 1994)). Intelligent control, in addition to the methods of adaptation and restructuring, uses such methods as *expert control* (Åström and Årzén, 1992), *neural* and *fuzzy control* (Berenji, 1992), *hybrid control* (Nerode and Kohn, 1993) and *learning control* (Kokar, 1993; Reveliotis and Kokar, 1995).

The goal of research presented in this paper was to study a new software paradigm – *adaptive software*. The structure of an adaptive program is patterned upon the structure of an adaptive controller. Towards this aim, we implemented a domain-specific (object/target recognition) program (we call it *RAACR: Reconfigurable Architecture for Adapting to Changes in the Requirements*) that incorporates some of the ideas of adaptive and reconfigurable control to deal with changes in specifications of software requirements.

The most typical meaning of the term "software requirements" is the functionality of the software, i.e., how it should respond to particular inputs from its "environment". But, as Zave and Jackson state (Zave and Jackson, 1997), *all* statements made in the course of requirements engineering are the statements about the environment. Inputs and outputs of a program are parts of the environment of the problem and any changes occurring at the inputs and outputs relate to the changes at the requirements of the program. This point of view is also reflected in the formal approach to software specification where the specifications of the functionality are expressed in terms of pre- and post-conditions. The pre-conditions specify what relationships should be fulfilled by the program inputs for the program to work correctly. Therefore, any changes in the inputs to the program that end up in the violation of the pre-conditions must be considered as changes in the software requirements. For this reason, in our study of adaptability of software to changes in the specifications we considered changes in the inputs to the program.

The changes in the inputs from the environment in our case study include: binary vs. gray scale images, noisy vs. noise-free images, types of objects in the images, alignment of edges with image borders vs. unconstrained orientation of edges. Since these specific changes in the environment were not represented in the initial specifications and were not considered in the design of our program, they represent the unexpected changes in our requirements specification.

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 in their processing. Consequently, the mechanisms we investigate in this paper are generic mechanisms (architectural solutions and statistical algorithms) rather than specific to the domain of object recognition. Our main thrust is to find an architectural solution that allows for the passing of feedback among particular system elements and to allow for the restructurability of the processing through changing the sequencing of the processing algorithms. Additionally, we add the processing capability (redundancy) required by the adaptive and restructurable control paradigms.

Our object recognition application, similarly as many other applications, is composed of different tasks, each accomplished in a different software component. 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 hierarchical multi-level blackboard (Shaw and Garlan, 1996; Erman et al., 1980; Nii, 1986).

The main reason for this kind of hierarchical organization is to reduce the complexity of processing. For example, object recognition on raw data, although possible, cannot 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. Another reason for the hierarchical organization is to increase the reuse of this kind of programs. Such abstraction layers containing functionality common to many applications in a specific domain, can constitute a *framework* (Baumer et al., 1997; Fayad and Schmidt, 1997; Posnak et al., 1997) that can be easily reconfigured (off-line) to a new application. For example, typical functionality for the image processing domain contains various filters, matrix operations and feature extractors that can be reused by various image processing programs.

The components of our application are called *knowledge sources (KSs)*; each of them implements an application-specific function. These components are organized into 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 structures and processing parameters of the KSs.

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 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 feedback obtained from other KSs in either the same or a different domain of the RAACR. This results in a real-time restructuring of the processing.

A system with multiple abstraction levels may consist of many KSs implemented as processes in each layer, running sequentially or parallel, communicating, cooperating or competing. To enable communication within the levels and among the levels, we implemented a *software bus*. In order to provide a generic (domain independent) bus for communication 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 (Obj, 1993). The system is implemented on the CORBA compliant SPRING operating system (Hamilton and Kougiouris, 1993).

To quantitatively evaluate the ability of the program to adapt to changing requirements, the *volatility of requirements* is measured using the *Total Requirements Volatility (TRV)* measure (Costello, 1997). The performance of the program for particular volatility levels is measured in terms of the probability of correct recognition. The dependence of the program's performance on the volatility of the requirements is obtained for both with and without the adaptation mechanism. Changing requirements are realized by providing inputs (images) that are out of bounds of the initial requirements specifications.

The paper is structured as follows. Section 2 briefly describes the RAACR and the main components of the architecture. This is followed by the description of the adaptation and reconfiguration mechanisms. In Section 3, an experimental scenario is described. Section 4 describes the evaluation steps and results of the experiments. And finally, in Section 5, the results of the experiments are analyzed and conclusions are presented.

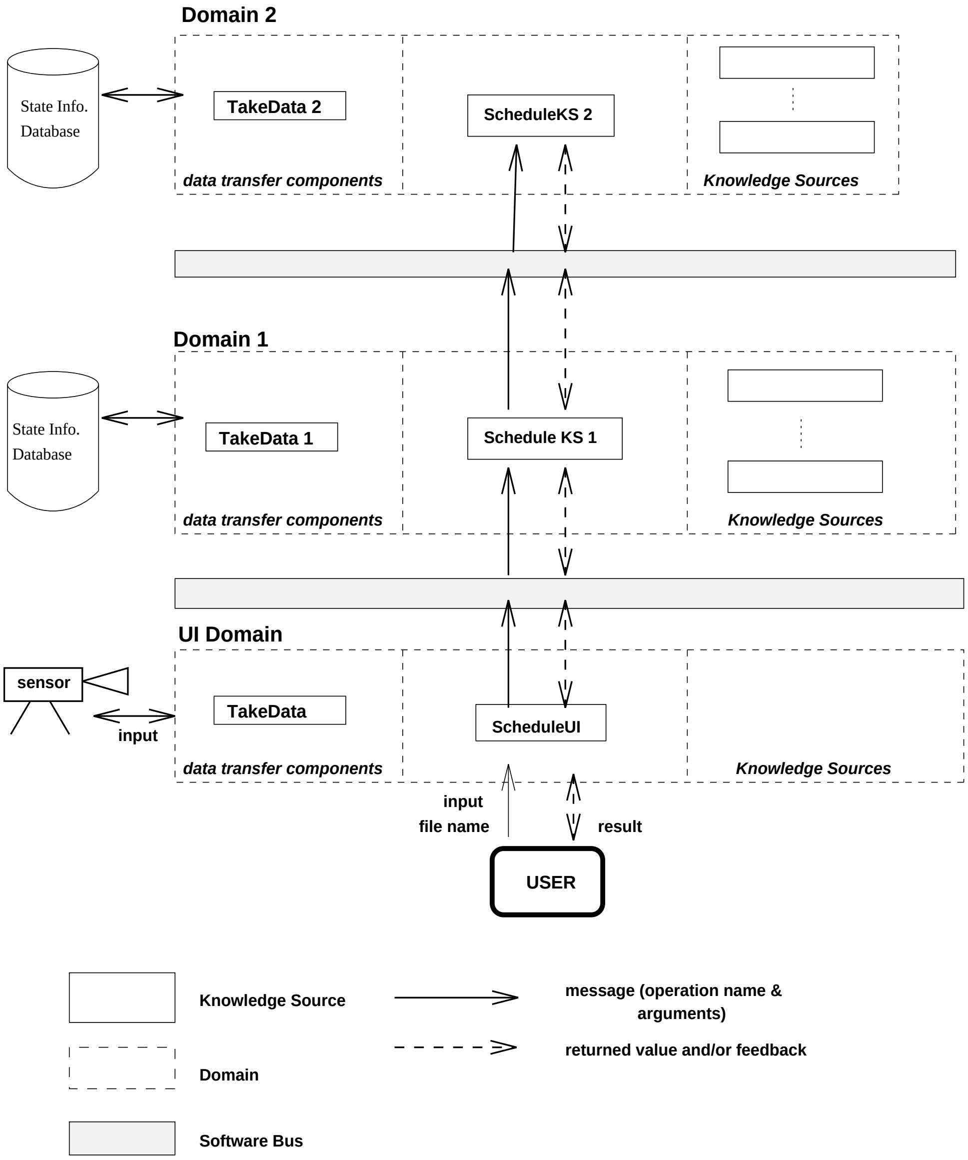

Figure 1: Top-level view of the RAACR.

2 RAACR: A Reconfigurable Architecture for Adapting to Changes in the Requirements

The top-level view of the RAACR is shown in Figure 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 subcomponents (shown as boxes inside of the domain representation): *data transfer components (DTC)*, *domain schedulers (DS)* and *knowledge sources (KS)*. Knowledge sources are the main working modules of an application. 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. DTCs provide the transfer of data between the domains and the databases, and among the domains. Additionally, they serve the purpose of initializing domains' data structures.

Knowledge sources are subcomponents of domains; they implement the main functionality of the application. According to the control terminology, we call this functionality *plant*. The adaptive control scheme (Åström, 1989) is represented in Figure 2. To implement such an adaptation scheme, a KS must have some additional functionality (*Model Estimator, Controller Designer, Controller*) and a number of additional attributes: *feedback*, *model-parameters*, *controller-parameters*, *control-input* and *goal*.

According to the adaptive control scheme, *Plant* receives *control-input* from *Controller*. This input affects the *Plant's* processing scheme. *Plant's* output (*feedback*) is sent to both *Controller* and *Model Estimator*. *Controller* calculates *control-input* according to its *control law*. *Control law*'s parameters are updated by *Controller Designer* based on the input from *Model Estimator*. *Model Estimator* estimates parameters of the *Plant*'s *model*, based upon both *control input* and *Plant*'s *output*, and passes the updated values to *Controller Designer*. All the parameters (model parameters, control parameters, goal) are initialized with a predetermined set of values stored in *State Information Database* of each domain (see Figure 1).

The restructuring process is represented in Figure 3. The main idea of this control scheme is to monitor the quality of control (*Model Selector*) and switch to

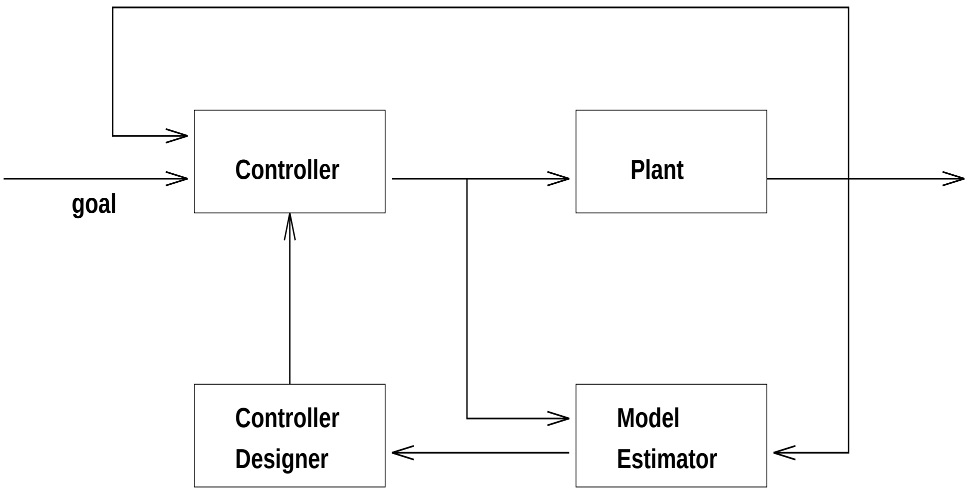

Figure 2: Adaptive Control Architecture

a new model and a new control law (*Controller Selector*) when the current control law does not provide satisfactory results.

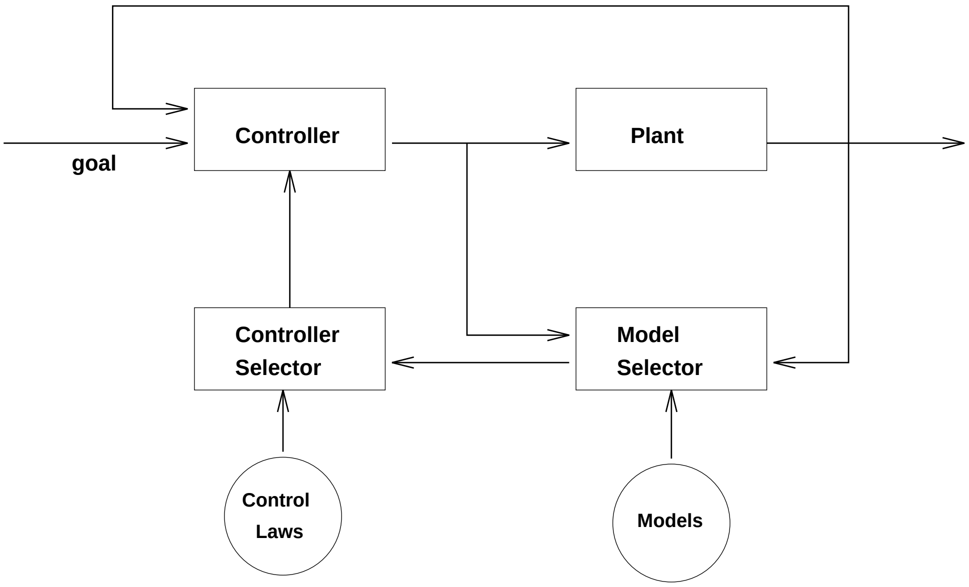

Figure 3: Restructurable Control Architecture

In RAACR, the restructuring scheme consists of the *Knowledge Source selection* mechanism implemented in the *domain scheduler* (*ScheduleKS* shown in Figure 1). *ScheduleKS* picks the best output of all the object operations performing the same kind of function. For instance, in the case of the edge detection application, two of the operations performing edge detection are *SobelEdgeFinder* and *Lapla-*

cianEdgeFinder. 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 better, since it is a directionless operator. *ScheduleKS* changes its selection procedure based upon the best output among the outputs of the operations performing the same kind of function based upon the feedback it receives.

3 Scenario

3.1 Object Recognition System Overview

To investigate the adaptability of the system to changing software requirements we use the application of object recognition. The input to the system (according to initial requirements) is a binary image containing one object. The goal of the system is to recognize whether the image contains a square or not. The system is shown in Figure 4.

The object recognition algorithm consists of two major processing steps: detecting edges of the object and recognizing the object. These two processing steps are implemented as two domains: the *image domain*, where edge points are detected, and the *histogram domain*, where edges are classified, angles between the edges are analyzed, and the object is recognized. Additionally, the system contains the *user interface domain*.

The image domain contains two knowledge sources: *SobelEdgeFinder* and *LaplacianEdgeFinder*. Each of these knowledge sources returns a histogram of the image. The histogram domain contains two knowledge sources: *FindEdges* and *IdentifyObject*. The details of these algorithms can be found in (Eracar, 1996).

3.2 Adaptation Mechanism

As we mentioned earlier, each knowledge source is a small system with an autonomous adaptive controller containing all the components, as shown in Figure 2: *Plant, Model Estimator, Controller Designer*, and *Controller*. Below we describe each of these components for the knowledge source *IdentifyObject* (see Figure 5).

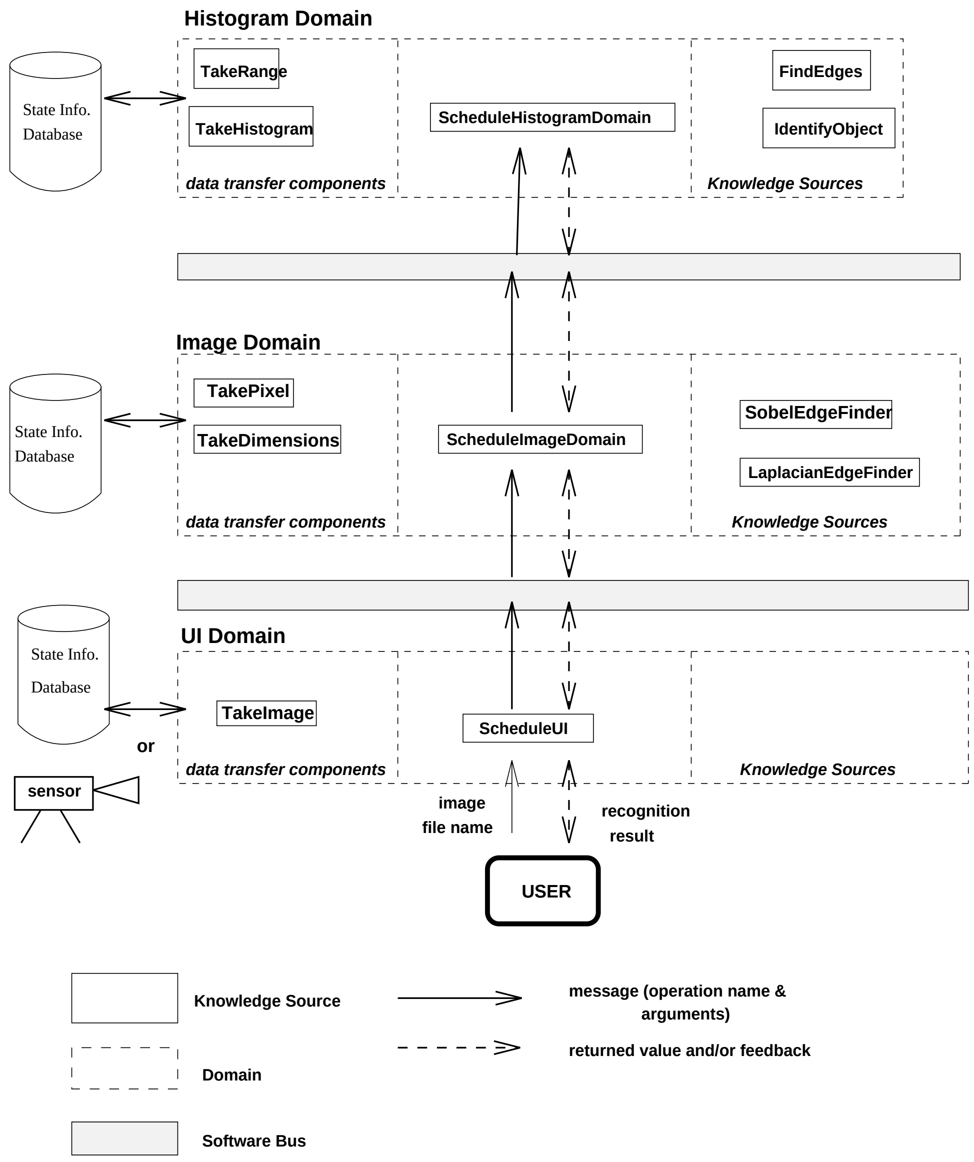

Figure 4: The domains of the experimental object recognition system.

The main function of *Plant* is to input a set of edge classes X, along with the environment feedback $\overline{\delta}(t-1)$ on the previous decision, and to produce a new decision $\delta(t)$: *square* or *non-square*. *Plant* makes this decision based upon a confidence constant α which is a control input from *Controller*. Essentially, *Plant* tests the hypothesis that the angles between consecutive edges are all 90 degrees. Additionally, *Plant* outputs some intermediate results to *Model Estimator*: $\overline{X_i}$, s_i and PCR, where $\overline{X_i}$ is the mean of the gradient direction for edge i, and s_i is its estimated variance. The details of calculating these values were presented in (Eracar, 1996). $PCR(t)$ is the feedback that goes to *Controller*; it is updated according to the feedback update equation

$$PCR(t) = \frac{PCR(t-1) \times (t-1) + \Delta}{t} \tag{1}$$

where

$$\Delta = \begin{cases} 0 & \text{if } \delta(t-1) \neq \overline{\delta}(t-1) \\ 1 & \text{if } \delta(t-1) = \overline{\delta}(t-1) \end{cases}$$

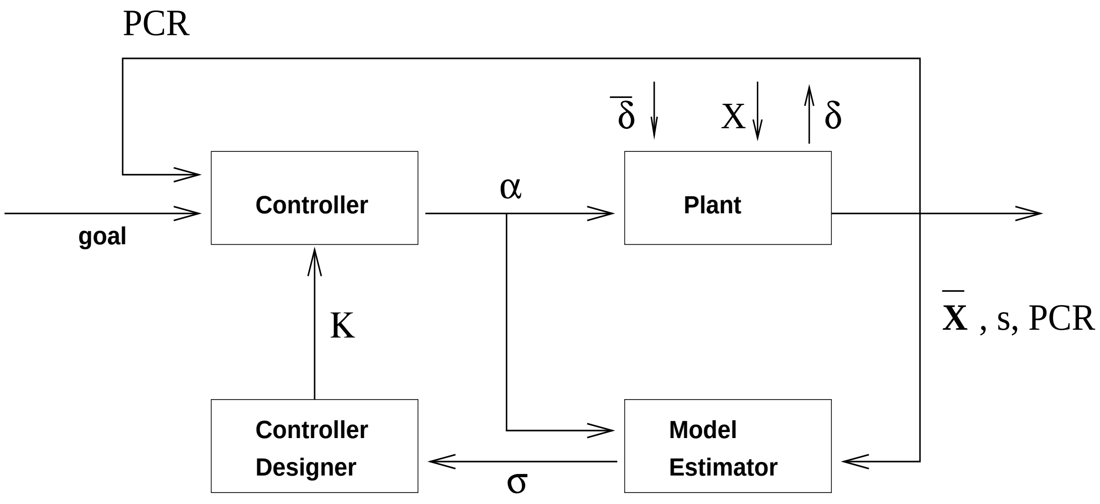

Figure 5: Adaptation Mechanism for IdentifyObject.

Model Estimator estimates the model of the plant. The model is probabilistic and 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 $i-1$. The model (estimates for μ_i and σ_i) is updated incrementally after receiving an input from the plant ($\overline{X_i}$ and s_i^2) according to the following rules:

$$\mu_i = \overline{X_i} - \overline{X_{i-1}} \tag{2}$$

$$\sigma_i = \sqrt{\frac{s_i^2}{n_i} + \frac{s_{i-1}^2}{n_{i-1}}}, \tag{3}$$

where n_i is the number of points in edge class i. The mean value σ of the updated estimates for σ_i is passed to *Controller Designer*.

Controller Designer updates the gain K of the control law according to the following equation:

$$K = C \cdot \sigma, \tag{4}$$

where C is a constant selected at the design time of the system.

The control goal is to achieve the high probability of correct recognition, i.e., to have $PCR(t) = 1$. Towards this goal, *Controller* sets the value of its output α according to the following *control law*:

$$\alpha(t) = K \cdot (PCR(t-1) - PCR(t)) + \alpha(t-1), \tag{5}$$

where $\alpha(0) = 0.05$. The new value of α is passed to *Plant* as the new control input.

3.3 Restructuring Mechanism for the Image Domain

In our application, the restructuring mechanism (see Figure 3) was used only in the *Image* domain. It restructures both the plant and the control functionality. Towards this aim, the domain scheduler (*ScheduleImageDomain*) makes a decision on which of the knowledge sources to schedule next, either *SobelEdgeFinder* or *LaplacianEdgeFinder*, based upon the feedback it receives from *IdentifyObject*. The feedback has the value of 1 if *IdentifyObject* is satisfied with the input provided to it and 0 otherwise. The feedback calculation is based on the number of edges. *IdentifyObject* expects a geometric object to have close to 4 edges. If the number of edges is greater than 5 or smaller than 3, the feedback value is 0, otherwise it is 1.

The restructuring policy is simply to select a different knowledge source in the list, when the previously received feedback was 0. If the feedback was 1, the selection is the same as before. The underlying presumption for this policy is that the input to the system has some inertia and sustains some pattern for a number of sequential images. Although this assumption is not always satisfied, it is a reasonable one; *general dynamic systems* (cf. (Mesarovic and Takahara, 1989; Klir, 1985)) are based on such an assumption. RAACR provides an architecture that can accommodate handling multiple knowledge sources and various restructuring policies. In this particular application, we had only two knowledge sources and this simple restructuring policy.

3.4 Measuring Changes in Requirements

To quantitatively evaluate the amount of changes in software requirements, the Total Requirements Volatility (TRV) metric was used (Costello, 1997). According to the methodology for calculating TRV described in (Costello, 1997), changes in the requirements must be simplified, divided and enumerated as independent countable pieces indexed by s. The time period has to be divided into time intervals enumerated by p. 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:

1. Add 1 if one new requirement was added, deleted, or modified.
2. Add $k+1$ if one requirement was split into k sub-requirements.
3. Add $k-1$ if k requirements are merged into one requirement.

Finally, the cumulative TRV measure, TRV_{cum}, is calculated according to the following formulas:

$$TRV_{cum} = RA_{cum} + RD_{cum} + RV_{cum} \tag{6}$$

$$RA_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RA(s,p) \quad (7)$$

$$RD_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RD(s,p) \quad (8)$$

$$RM_{cum} = \sum_{p=p_1}^{p_m} \sum_{s=s_1}^{s_n} RM(s,p) \quad (9)$$

In our study we entertained an idea of using Function Points (Albrecht, 1979; IFP, 1994) to measure initial requirements, instead of Total Requirements Volatility measure. In the Function Points method, the visible aspects of the software system are examined: inputs, outputs, inquiries, data files, and interfaces. These five aspects are counted, multiplied by an appropriate weight and added together; this results in an *Unadjusted Total*. The Unadjusted Total is multiplied by a *Value Adjustment Factor* (VAF), which depends on 14 general system characteristics. The items are weighted according to the difficulty of their implementation. The weighting factors were developed empirically (IFP, 1994).

While the idea of using Function Points seemed to be a natural choice, it is not appropriate for our problem, mainly because the aspects that define the Function Points measure do not reflect the changes in the environment. The number of inputs, outputs, inquires, data files and interfaces in each of the modules in our application remains constant and does not relate to the fact whether the module needs to deal with the noise or noise-free input, whether the objects to be identified are aligned with the frame borders or not. Simply stated, the Function Points measure is not sensitive to the changes in the environment and thus cannot be used to evaluate adaptive processing software. For this reason, the TRV measure, although less known in the literature, was more appropriate for our task.

3.5 Measuring Performance

To evaluate the degree of degradation of the system performance that results from changes in the requirements we use the probability of correct recognition ($P(p)$) of the object. This probability was calculated as:

$$P(p) = \frac{q+n}{N}, \quad (10)$$

where q is the number of squares recognized correctly, n - number of non-squares classified as non-squares, N - number of images for the time period p.

3.6 Incremental Changes

The main idea behind our study 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 TRV measure. The four types of input images are described 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 bases of triangles 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. In addition to the images as above, Shot and Gaussian noise is added to the images of Set 3. Images can be of the gray scale type.

According to the methodology described in (Costello, 1994), these 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. $S(4,1)$. Images can contain Gaussian noise.

9. $S(4,2)$. Images can contain Shot noise.

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

These requirements are shown in Table 1. The associated values of the measures RA, RD, RM and TRV are shown in Table 2.

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	$S(3,1)$	$S(1,2)$(deleted)
4	$S(4,1)$,$S(4,2)$,$S(4,3)$	$S(1,3)$(deleted),$S(1,4)$(deleted)

Table 1: **Requirements changes for each image set.**

Set:	RA	RA_{cum}	RD	RD_{cum}	RM	RM_{cum}	TRV_{cum}
1(base)	0	0	0	0	0	0	0
2	2	2	0	0	1	1	3
3	1	3	1	1	0	1	5
4	3	6	2	3	0	1	10

Table 2: TRV **values for each image set.**

4 Experimental Results

To evaluate the adaptability of the system we analyzed the effect of change in the software requirements on the system performance. Towards this aim, we measured the probability of correct recognition for each of the four levels of change as measured by the TRV measure. We have also compared the performance of the RAACR against the system without the adaptability mechanism. For this purpose, we ran the same set of experiments with the adaptation mechanism turned off. The same sets of input images were input to this non-adaptive system in the same order as for the adaptive system and the probability of correct recognition $P(p)$ was calculated.

The results of our experiments are summarized in Figures 6, 7, and 8. Figure 6 shows the probability of correctly recognizing squares (dark bars), and probability of correctly recognizing non-squares (white bars), for each measured level of the TRV measure. The same kind of results are shown in Figure 7 for the non-adaptive system. Finally, in Figure 8, the probabilities of correct recognition, $P(p)$, as defined by Eq. 10, for the two cases are compared.

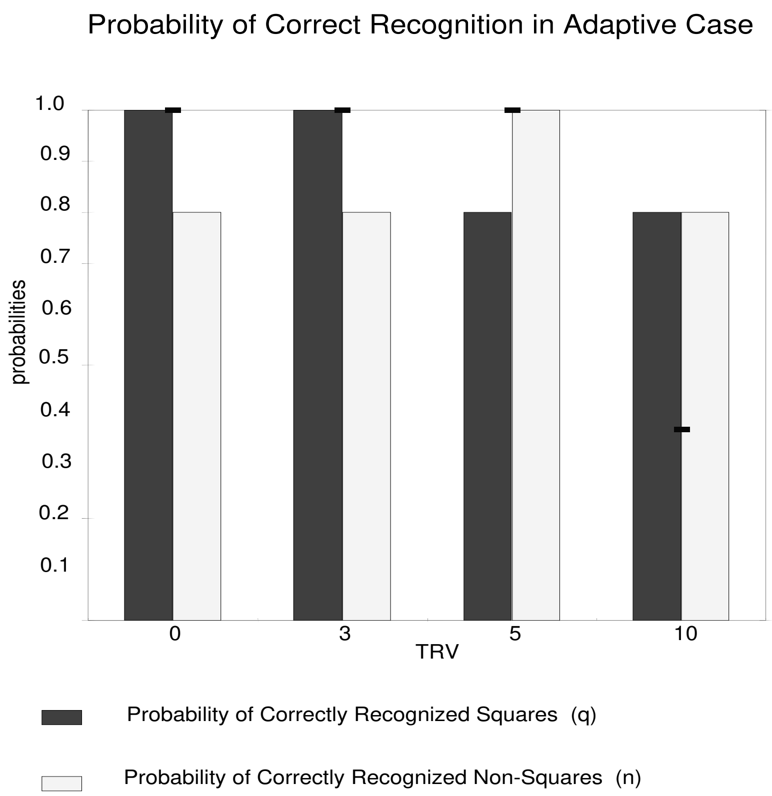

Figure 6: Probability of Correct Recognition for the RAACR

5 Conclusions and Future Research

Control and adaptation embedded into algorithms are not new to software systems. Examples of software that incorporates some kind of adaptation include the

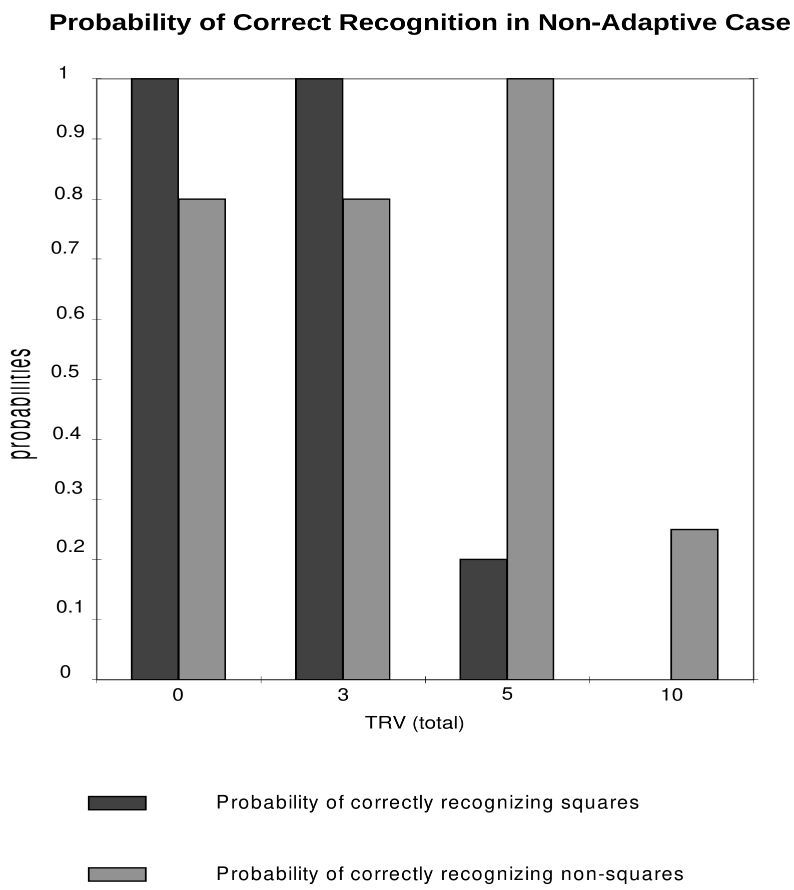

Figure 7: Probability of Correct Recognition for Non-Adaptive System

following:

- Software for dynamic adjustments of the buffering strategy in a database management system;
- Routing algorithms for networks;
- Load balancing algorithms for distributed computer systems;
- Graphical User Interfaces (GUI) that adapt to a specific user;
- Caching strategies for memory management in operating systems.

Our research goal is to develop a methodology for designing adaptive software that would (1) unify the design of adaptive software for different application domains, and (2) extend the capability to adapt to other domains for which such capabilities have not been established. In this paper we addressed the issue of architecture for such adaptive systems. Towards this goal, we achieved the following results.

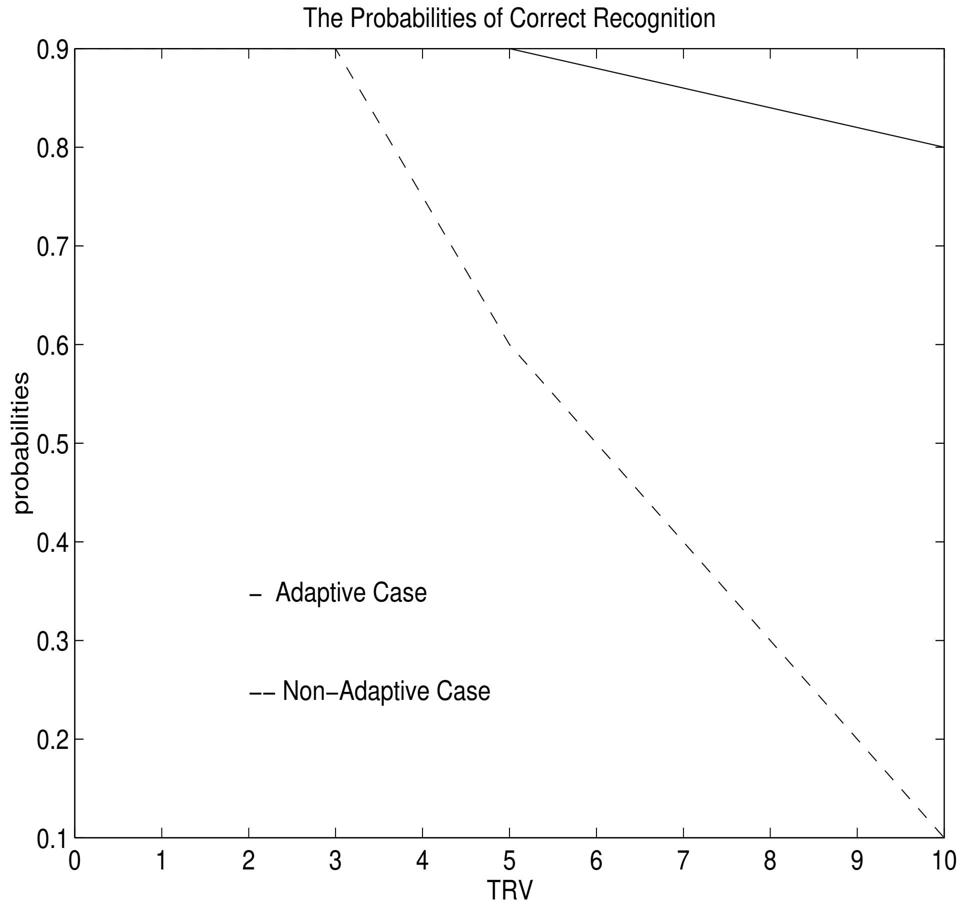

Figure 8: Comparison of the Probabilities of Correct Recognition of Adaptive and Non-Adaptive Systems

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

2. We investigated a number of features that can be used in a framework for building software systems that are adaptable to changes in the software requirements through adaptation and restructuring of control. The main features of this framework 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) Closed-loop organization of the main functionality of the application (plant) and the feedback processing mechanisms. The feedback processing mechanisms include: calculation of feedback, model estimation, controller design and control.

(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 and knowledge sources.

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 TRV 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 kind of capability does not come for free. As shown in this paper, the functionality of the application needs to be supplemented with some redundancy to implement the mechanisms of control: feedback, adaptation and restructuring. It also should be noted that the results shown in this paper constitute an initial step in the right direction. To achieve the level of knowledge that would allow for engineering of systems adaptable to changes in software requirements, more research is necessary. This includes such issues as: generic feedback calculation mechanisms (for instance, the Quality of Service (QoS) metrics (Parris et al., 1993; Sabata et al., 1997)), generic adaptation and restructuring mechanisms, efficient communication mechanisms, support for measuring changes in requirements, and many others.

Another issue that needs to be investigated is the scope of applicability of such adaptation mechanisms to software engineering at large. The control technology has been developed for dynamical systems, i.e., systems that have *state* and change their state due to both external inputs and as a result of the passing of time. Such systems exhibit some *inertia*, i.e., their behavior pattern does not change instantly, it changes slowly over time, a feature that is important for the adaptability of the controller. The question is which of the software systems can be classified into this category and how large is the percentage of such systems among all software systems?

The architecture described in this paper has been used in the design of two other programs: a spell checker and a local area network analyzer. We briefly describe these programs simply to show that the same ideas and the same architecture are applicable in various programs and that they are not limited to the image processing domain.

A typical spell checker consists of two steps: check each word against a dictionary, and if a word is not found in the dictionary, suggest to the user what should be the correct word. The latter step involves finding a close match between the word that was flagged as misspelled and a word in the dictionary. The result of such a matching process is not unique and thus the spell checker can give a number of suggestions. Too many suggestions could create a problem, since the user would need review the list each time such a problem is identified. In a non-adaptive system, if the user makes the same mistake the system always gives the same suggestions and in the same order. An adaptive system, on the other hand, can adapt to the user and give suggestions that reflect the typical behavior for that particular user. Such an adaptation process is driven by the feedback, which is part of normal interaction with the spell checker, i.e., from the selections the user makes whenever a specific list of suggestions is displayed. Our adaptive spell checker has five KSs related to particular user mistakes: Left-Right Character Shifter, Character Doubler, End Character Appender, Character Remover and Subsequent Character Switcher. Each of the KSs makes a number of suggestions; all of them are passed to the evaluation domain. Also, probabilities of correct suggestion are maintained by the KSs and passed along with the suggestion to the evaluation domain. Evaluation domain checks suggestions against a dictionary and passes valid suggestions to the display. The valid suggestions are displayed in the order that reflects the current knowledge about the given user's probability of particular mistakes. The most likely mistakes (suggestions to correct them) are displayed at the top of the list. After the user makes a selection, feedback is passed back to the KSs, indicating whether their suggestions were correct or not. In response to the feedback, the KSs update their probabilities of correct suggestion. As a result of this adaptation mechanism, the most likely suggestions are shown to the user first.

The LAN analyzer is designed to diagnose errors on a computer TCP/IP network and to display error messages to the network administrator. The idea is not to display a message related to the same fault over and over again. The goal is to keep the number of error messages within some bounds. Too many messages will overwhelm the network administrator, while missed errors will result in the network failure.

We focused on four kinds of errors, designing a knowledge source for each type of error: ping failure, TCP connect failure, repeating overload, and node or network down. The goal for the system is to inform the operator of a particular error only a limited number of times. Also, the system should be able to adjust its threshold on the number of repeated packets with failures. The system is organized into three domains: input domain (collecting raw data from a snoop process), symptom domain (four KSs responsible for detecting particular errors), diagnosis domain (an evaluation KS that processes feedback from the network administrator and two KSs that process that feedback and generate feedback for particular knowledge sources of the symptom domain). The system, through the interaction with the administrator, adjusts its behavior so that the administrator is not overburdened with too many error messages and thus is more free to make decisions regarding the network performance.

References

Albrecht, A. (1979). Measuring application development productivity, *The Joint SHARE/GUIDE/IBM Application Development Symposium*, pp. 83–92.

Antsaklis, P. (1994). Defining intelligent control: Report of the task force on intelligent control, *IEEE Control Systems Magazine* **14, No.3**: 4–5, 58–66.

Åström, K. J. (1989). *Adaptive Control*, Addison-Wesley, Reading, MA.

Åström, K. J. and Årzén, K.-E. (1992). Expert control, *in* K. M. Passino and P. J. Antsaklis (eds), *Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers.

Baumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Zullighoven, H. (1997). Framework development for large systems, *Communications of the ACM* **40**(10): 43–47.

Berenji, H. R. (1992). Fuzzy and neural control, *in* K. M. Passino and P. J. Antsaklis (eds), *Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers.

Costello, R. (1994). *Metrics for Requirements Engineering*, Master's thesis, California State University, Long Beach.

Costello, R. (1997). Requirements tracibility metrics for the software system lifecycle, *Applications of Software Measurement Conference*, Software Quality Engineering, Atlanta, Georgia.

Eracar, Y. (1996). *RAACR: A Reconfigurable Architecture for Adapting to Changes in the Requirements*, Master's thesis, Northeastern University, Boston, MA.

Erman, L., Hayes-Roth, F., Lesser, V. and Reddy, D. (1980). The Hearsay-ii Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, *Computing Surveys* **12**(2): 213–253.

Fayad, M. and Schmidt, D. (1997). Object-oriented application frameworks, *Communications of the ACM* **40**(10): 32–38.

Hamilton, G. and Kougiouris, P. (1993). The spring nucleus: A microkernel for objects, *Proc. of the 1993 Summer Usenix Conference*, Cincinnati.

IFP (1994). *Function Point Counting Practices Manual, Release 4.0.*

Klir, G. (1985). *Architecture of Systems Problem Solving*, Plenum Press.

Kokar, M. M. (1993). Learning control: Methods, needs and architectures, *in* K. M. Passino and P. J. Antsaklis (eds), *Introduction to Intelligent and Autonomous Control*, Kluwer Academic Publishers, pp. 263–282.

Mesarovic, M. D. and Takahara, Y. (1989). *Abstract Systems Theory*, Springer Verlag.

Montoya, R. J., Howell, W. E., Bundick, W. T., Ostroff, A. J., Hueschen, R. M. and Belcastro, C. M. (1982). Restructurable control, *Technical Report NASA CP-2277*, NASA Langley Research Center, VA.

Nerode, A. and Kohn, W. (1993). *Hybrid Systems*, Springer Verlag.

Nii, P. (1986). Blackboard systems, *AI Magazine* **7**(3): 38–53.

Obj (1993). *The Common Object Request Broker: Architecture and Specification*, revision 1.1 edn.

Parris, C., Ventre, G. and Zhang, H. (1993). Graceful adaptation of guaranteed performance service connections, *Globecom'93*, Houston,TX.

Posnak, E., Lavender, G. and Vin, H. (1997). An adaptive framework for developing multimedia software components, *Communications of the ACM* **40**(10): 43–47.

Reveliotis, S. A. and Kokar, M. M. (1995). A framework for on-line learning of plant models and control policies for restructurable control, *IEEE Transactions on Systems, Man and Cybernetics* **25, no. 11**: 1502–1512.

Sabata, B., Chatterjee, S., Sydir, J. and Lawrance, T. (1997). Hierarchical modeling of systems for QoS based distributed resource management, *Technical report*, SRI International.

Shamma, J. S. (1996). Linearization and gain-scheduling, *The Control Handbook*, CRC Press.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Zave, P. and Jackson, M. (1997). Four dark corners of requirements engineering, *ACM Transactions on Software Engineering and Methodology* **6**(1).