

An Architectural Framework for the Design and Analysis of Autonomous Adaptive Systems

Kendra Cooper João W. Cangussu Eric Wong
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688, USA
{kcooper,cangussu,ewong}@utdallas.edu

Abstract

Autonomous adaptive systems (AAS) have been proposed as a solution to effectively (re)design software so that it can respond to changes in execution environments, without human intervention. In the software engineering community, alternative approaches to the design of AAS have been proposed including solutions based on component technology, design patterns, and resource allocation techniques. A key limitation of the currently available approaches is that they detect constraint violations, but they do not support the prediction of constraint violations. In this work we propose an Architectural Framework for the Design and Analysis of Autonomous Adaptive Systems, hereafter referred to as KAROO, which provides a key, new contribution: the capability to predict when a system needs to adapt itself. The results of extensive experimental evaluation of a KAROO-based system are excellent: 100% of the violations are predicted; the system is able to avoid the violations by adapting itself almost 98% of the time. The framework is a novel integration of control-theory-based adaptation, multi-criteria decision making and component-based software engineering techniques.

1 Introduction

The environments in which software are executing today have increased considerably in complexity, and software may need to adapt themselves at run-time to changes in the security level, CPU utilization, memory usage, etc. These self adapting systems present challenging new (re)design issues. As a system's behavior is non-deterministic, the execution environment must be monitored; changes to the system (e.g., to avoid a memory constraint violation) must be reasoned about to predict violations in advance. This gives the system time to adapt before the violation occurs.

The system also needs to identify the necessary changes that would best meet multiple, often conflicting, quality of service (QoS) objectives. The capabilities to monitor, predict, and self-adapt need to be (re)designed into new and existing legacy systems. This is emerging as a key research problem in software engineering: “How can we effectively (re)design software so that it can respond to changes in execution environments, without human intervention?”

Recently, *autonomous adaptive systems* (AAS) have been proposed as a solution to this problem. An AAS is one in which the system adapts itself at run-time to respond to changes in the environment [9]. In the software engineering community, a number of approaches to the design of AAS have focused on solutions based on component technology [15] and design driven approaches using patterns [6, 8, 12]. Alternative approaches have focused on resource allocation, such as an operating system scheduling processes [18], QoS guarantee [12], and fault tolerance. Adaptive frameworks for agent-based [19] and grid-based [22] systems have also been proposed. A key limitation of the currently available approaches is that they detect constraint violations, but they do not support the prediction of constraint violations. In addition, changes to external conditions such as security levels have not been fully addressed in the literature.

Here, we propose an Architectural Framework for the Design and Analysis of Autonomous Adaptive Systems, hereafter referred to as *KAROO*¹, which addresses the dynamic adaptation issues described above. The *KAROO* Framework is a novel integration of control-theory-based adaptation, multi-criteria decision making and component-based software engineering techniques; it provides a general framework for run-time adaptation. In addition to the advantages of many adaptive system approaches (self-

¹Karoo is a chameleon species and also the name of a region in South Africa where fauna and flora are impressively adapted to the extreme climate.

adaptation, robustness, better resource usage, etc.), the *KAROO* Framework has three distinguishing contributions: flexible usage, predictable adaptation, and automatic reasoning. The combination of these key features provides a novel, innovative solution to the complex problem of (re)designing AAS. The *KAROO* Framework has been validated by implementing a prototype of a *KAROO*-based application, which provides a data persistency service. Using an extensive experiment, the correctness of the system in identifying violation constraints and the ability to adapt the system before the violation occurs are measured.

The remainder of this paper is organized as follows. Section 2 presents a summary of existing approaches for AAS. A description of the proposed approach can be found in Section 3. The results of using a prototype of the *KAROO* framework to a remote data persistency service are presented in Section 4. Conclusions are drawn in Section 5.

2 Related Work

Autonomous adaptable systems are needed in a wide variety of domains. Unmanned autonomous vehicles are needed in remote and/or hazardous conditions in submersible, aerospace, and military/police systems. Autonomous adaptable systems for webservers and wireless multimedia have been proposed in addition to collaborative learning environments [2], medical [21], and manufacturing applications. The approaches to develop such systems are also diverse, including biologically inspired approaches (e.g., based on immune or neurological systems), agent-based systems [19], scheduling techniques [7, 10], component technology [15], grid-based systems [22] and design driven approaches [8, 14]. Many approaches are based on the sound theory of feedback control [1, 12].

Here, we survey design driven approaches, as these are the most closely related to our research. The vast majority of design driven approaches in the literature are related to engineering new autonomous adaptable systems.

The architecture of autonomous adaptable systems has been considered using a variety of layered and process control approaches. The architectures share three common elements: provide mechanisms to monitor the execution environment, identify the need for adapting the system, and realize the adaptation of the system. Control theory has been applied to the design of autonomous systems that adapt resources, such as CPU scheduling, network bandwidth, queue management, etc., to maintain QoS guarantees [1, 12].

2.1 Process Control Architectures

An architecture is proposed in [21] to address scalability issues in large, event driven, distributed systems using tradi-

tional feedback control theory. Scaling event processing requires an architecture that incorporates parallelism and provides flow control within replicated elements to avoid overload conditions and load balancing in the presence of variable processing demands. The architecture has two tiers. The first tier is responsible for intra-event processing, such as filtering events from specific sources. It considers events in isolation and can be readily parallelized by replicating the processing elements. The second tier is responsible for inter-event processing, such as problem diagnosis; multiple events are analyzed in combination. Control theory is used to provide flow control and load balancing mechanisms in the first tier. For example, the flow control system seeks to maximize throughput without dropping events by regulating the free space of the result queue. The reference input is the desired free space; the controller uses the difference between this reference and the measured free space of the result queue to adjust the event input rate. The use of traditional feedback control theory has some limitations, however, in that it requires additional profiling, identification, tuning, etc., in order to produce the desired QoS on each new platform.

The Adaptive Control Framework for QoS project is based on adaptive control theory [12]. The key advantage of adaptive control is that it can automatically re-tune the controller in response to changes in the system model; this makes the services more QoS-portable because they automatically tune themselves to the platform they are installed on. Here, the controlled system is a differentiated caching service; the actuator in the example is the disk space manager. The parameter estimation and controller design process are automated in this work. The approach has been designed to detect violations, but not to predict them.

2.2 Layered Architectures

A layer-based approach to constructing adaptive software in distributed systems is presented in [6]. Here, the software executing on each host is organized into layers (application, middleware, operating system, and network); each layer can be either adaptive or non-adaptive. An adaptive layer is composed of a collection of adaptive components. A collection of adaptive components (located on multiple machines) cooperate to provide part of a distributed service. Each adaptive component is composed of two different kinds of modules. The first is a component adapter module, which controls the component's adaptive behavior. The second is an alternative adaptation aware algorithm module; each of these provides a different algorithm that implements the functionality of the component. When the component adapter module perceives that a different algorithm is a better fit for the requirements, it coordinates the change, including inter-host communication

and state transfer between algorithm modules. In addition to the layers, an adaptive controller subsystem is responsible for coordinating inter-layer and inter-component adaptations on each host. The architecture has been designed to detect violations that occur, but not to predict them.

The Rainbow architectural framework proposes a reusable, layered approach to capture the static and dynamic behavior of autonomous adaptable systems [8]. The first layer is the System-layer infrastructure, which provides a system measurement mechanism, realized as probes to observe and measure various states of the system, a resource discovery mechanism that can be queried for new resources, and an effector mechanism that can carry out the modifications. The second layer is the Architecture-layer infrastructure, which is responsible for aggregating information from the probes and updating the appropriate properties in the model; an adaptation is triggered if a constraint violation is discovered. An adaptation engine determines the course of action and carries out the changes. Architectural styles [14] augmented with dynamic attributes are proposed to reuse knowledge effectively across systems in similar domains. The Architecture-layer infrastructure has been designed to detect violations that occur, but not to predict them.

3 Architectural Framework

After carefully reviewing the capabilities that need to be realized in the architecture, the Feedback Control style [17] is selected to provide the overall structure and behavior (refer to Figure 1). The purpose of a control system is to “maintain specified properties of the outputs of the process at (sufficiently near) given reference values called the set points” [17]. We note here that a process control style can be used for systems in which sensors provide data either continuously or discretely [13]. The Feedback Control style accomplishes three fundamental things: (i) Monitor process variables related to QoS properties of interest including system (e.g., memory usage, response time performance, etc.) and environmental (e.g., security levels, etc.); (ii) Predict the violation of a set point (the desired value for a quality of service variable); and (iii) Control the application’s behavior by replacing one or more components at run-time. The Structural and the Dynamic view of the architecture are described in Sections 3.1 and 3.2, respectively.

3.1 Structural View of the Architecture

This style has two main subsystems: the process and the controller. Here, the process is a domain specific application such as a medical image diagnostic system, e-commerce application, etc.; the controller is the domain independent adaptation mechanism provided in the framework. The interactions support the coordinated replacement

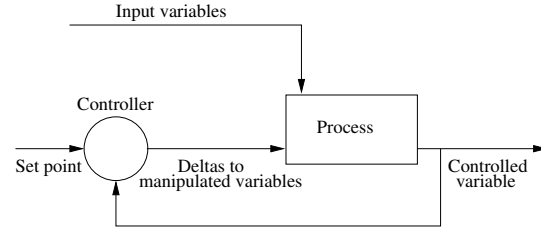


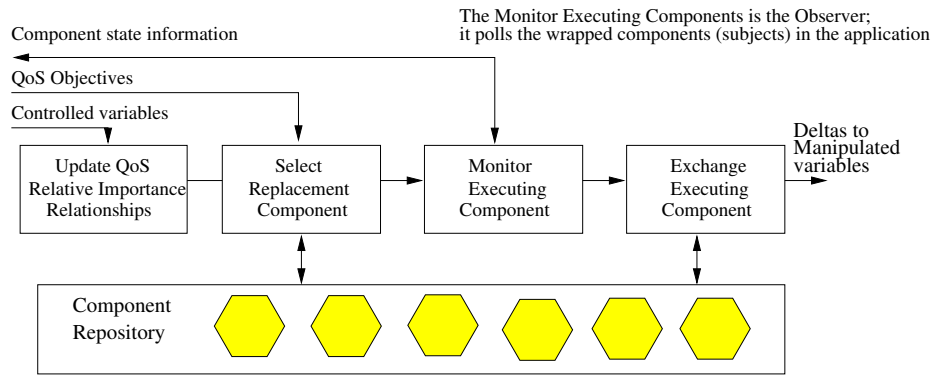
Figure 1. Overall Architecture is based on the Feedback Control Style.

of components at run-time to achieve a desired QoS.

The process, or application, subsystem can be structured using established patterns (e.g., layered, model view controller, blackboard, etc.) [4, 17]. Here, the application is a component-based application, which means that off-the-shelf components are used rather than building the entire application from scratch. The use of components holds the promise of supporting the timely, cost effective development of large-scale, complex systems [3]. The components can provide functional and non-functional capabilities; they have already gone through the time consuming development activities of requirements specification, design, implementation, and verification. The use of components is viewed as a means to effectively meet the needs of rapidly changing business environments. In this work, we do not restrict the components to off-the-shelf (OTS) components, or components for which the source code is available. Our approach supports the use of commercial off-the-shelf (COTS) components, where only the externally visible, blackbox behavior of the component can be obtained and specified; we do not assume that source code meta-data is available to support a reflective approach.

Figure 2 illustrates the processor subsystem and the wrapped components. The adaptation of the system is achieved by exchanging components in the application at run-time. To accomplish this, wrappers are used to collect information about the executing components. As a “hot swap” of COTS components in general is not feasible, the component must stop execution before it can be replaced. In addition to collecting information, the wrappers also provide a consistent component interface for the application. Each wrapped components is a Subject instance in an Observer pattern; the observers are located in the controller subsystem.

The first level of decomposition of the Controller Subsystem is presented in Figure 3. The Resource Monitor subsystem is responsible for collecting data about the input variables that are related to the QoS properties. Here again, we use the Observer pattern to periodically collect data. In this subsystem, the state information collected includes system resources (e.g., how much memory is available in the



The Replace Component subsystem is part of a broker pattern, where the client is the application (processor) subsystem, server is the Component Repository, broker is the Update QoS Relative Importance Relationships, Select Replacement, Monitor Executing, and Exchange Executing Component subsystems.

Figure 4. Decomposition of the Replace Component Subsystem

system, what is the CPU utilization, what is the network bandwidth utilization, etc.) and environmental resources (e.g., current security level, etc.) The data are used by both the System Identification Module (SIM) and the Resource Analysis subsystems.

The System Identification Module (SIM) is responsible for creating and updating the system model. System identification deals with characterizing the input/output behavior of an unknown system from empirical data [11]. The rigorous determination of the parameters of the system is quite difficult; however, it is possible to estimate them using a variety of approaches. The least-squares method is a widely used approach. Here, the system equations are viewed as a set of equations which are unknown in system parameters and in which certain functions of measurements serve as regressors. A consistency measure, defined as the ratio of standard deviation of the estimates to their mean value, can be used to determine if the model of the system based on the mean values is in agreement with the actual system.

The Resource Analysis Module not only captures the values provided by the Resource Monitor but also uses the model(s) created by SIM to extrapolate the behavior of the resource of interest and predict any constraint violation specified by the “set points”.

The Replace Component subsystem is decomposed into the QoS Relative Importance Relationships, Select Replacement, Monitor Executing, and Exchange Executing Component subsystems (refer to Figure 4).

3.1.1 QoS Relative Importance Relationships

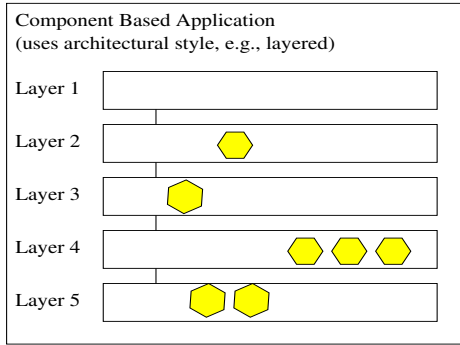
This subsystem provides the capability to initialize and dynamically update the QoS relative importance relationships. For example, in Figure 5, if the security level in the envi-

ronment changes from yellow to orange, then the relative importance relationships among the various QoS properties need to be updated to reflect the change (i.e., security is now more important than it used to be). This is accomplished using a set of rules that embody pairwise comparisons, using values from the AHP scale. Another example from Figure 5 is that if memory usage is strongly preferred with respect to security, then this relative importance is assigned the value 7.

These QoS relative importance relationships form the top level decomposition of a multi-criteria decision making approach. In traditional multi-criteria decision making approaches, the matrices for each decomposition level are static. In the work proposed here, dynamically updated matrices are used to reflect changes in the environment and system resources.

3.1.2 Select Replacement Component

This subsystem provides the capability to select a replacement component. This is a multi-criteria decision making problem: the components each have their own QoS behavior (e.g., memory usage, response time, etc.), and a replacement component needs to be selected with respect to the current QoS objectives in the system. Consider a situation in which a currently executing component is very fast, but requires a large amount of memory; a prediction has been made (by the Predict Violation subsystem) that a memory usage violation is going to occur. The current QoS objectives would lead to the search for a component that provides the same capabilities, but uses a low or moderate amount of memory, in addition to other desired QoS properties. The components’ QoS behaviors are modeled using empirical data that are systematically collected. The experimental de-



Each COTS (blackbox) component is wrapped to create a Subject instance
 – collect/report information about its execution (part of the Observer pattern)

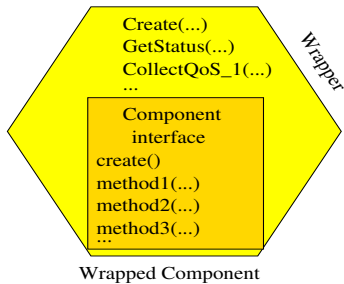


Figure 2. Decomposition of the Process Subsystem and the Wrapped Components

sign is available in [5]; over 10,000 data points are collected for each component.

There are numerous multi-criteria decision making approaches available. Here, we use the AHP [16] to select a component. We use a straightforward decomposition of the problem into three levels. The top layer represents the highest level objective; the second level represents the attributes such as memory, response time, security, data integrity, etc.; and the lowest level represents the alternative components that are available.

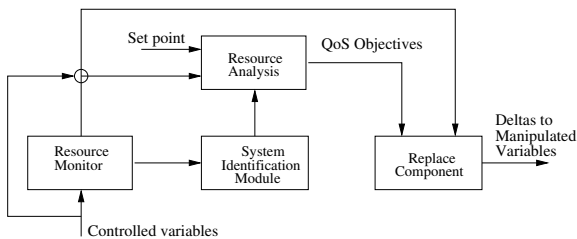


Figure 3. Decomposition of the Controller Subsystem

```

if security == yellow then
  memory usage:security = 7
  response time performance:security = 8
  data integrity= 7
  ...
if security == orange then
  memory usage:security = 5
  response time performance:security = 6
  data integrity= 8
  ...
if security == red then
  memory usage:security = 3
  response time performance:security = 4
  data integrity= 9
  ...
  
```

Figure 5. Example of rules to update the relative importance among QoS properties.

3.1.3 Monitor Executing Components

When exchanging components, the adaptation mechanism needs to determine when the components are in a “safe” state (a state which will not compromise the system and/or the application) to be replaced. Hence, the system needs the ability to monitor and report the state of executing components. This is achieved using the Observer pattern [4]. The Observer pattern helps to synchronize the state of cooperating design elements. The pattern defines subjects and observers; each subject can notify one or more observers about changes in its state. Here, we use the pattern to obtain state information from the components and determine when it is “safe” to replace one component with another. Each subject is a component (or collection of components) with a wrapper. The wrapper is used to collect state information about the component. These subjects can be located in any part of the application. They can be polled for information by the observer, which resides in the adaptation subsystem.

3.1.4 Exchange Components

The ability to exchange components in the application is achieved using a broker pattern [4]. The broker pattern allows an application to access distributed services using location-transparent service invocations. This pattern provides flexibility, as it allows the dynamic exchange, addition, and removal of components at run-time. The pattern definition includes clients, servers, and brokers. A client makes requests to access the services of one or more servers. The request is sent to a broker, which is responsible for the transmission of requests and responses between the client and server. A broker can register servers and invoke server methods. A server provides functionality, which is available via method invocations. Here, the client is the application subsystem, the server is a collection of components in a repository, and the broker is the Replace Component subsystem. When combined, the application capa-

bilities and the autonomous adaptation capabilities form a closed loop feedback control system.

3.2 Dynamic View of the Architecture

The dynamic view of the architecture is presented in Figure 6 using an activity diagram from the Unified Modeling Language. The activity diagram is well suited for capturing this view, as it is a behavioral diagram in which the structural subsystems (e.g., Resource Monitor, Resource Analysis, etc.) each have a swim lane in the diagram, which is a vertical slice of the figure. The sequence of the processing performed is represented using activities (rounded edge rectangles), objects for temporary data storage (rectangles), flows (directed arrows), and decision points (diamonds).

The processing begins with the Resource Monitor collecting data about the system and environmental resources. System resources include the amount of memory utilization, CPU utilization, network bandwidth utilization, etc. Environmental resources are elements which are outside the system, such as the security level (e.g., red, orange, etc.). The Resource Monitor periodically checks the resource variables of interest and performs error checking on the data. The data the Resource Monitor collects is used by the SIM, the Resource Analysis, and the Replace Component subsystems. The data collected by the Resource Monitor are accessed using a polling mechanism. SIM uses the data when it periodically updates the (mathematical) control system models.

The Resource Analysis subsystem activities begin by requesting the input data needed: the QoS variable data (from the Resource Monitor), the set points, stored as configuration data, and the SIM. The configuration data are not shown in the dynamic view to simplify the figure. The Resource Analyzer determines if the system needs to be adapted. The causes for adaptation include: a system resource constraint is currently violated, a system resource constraint is predicted to be violated, or changes in the environment require the system to change. If the system needs to be adapted, then the activities to replace components begin.

The Replace Component subsystem activities begin by updating the QoS relative importance relationships. After this is done, the system searches for new components to better suit the current environment using the AHP approach. The component specifications, derived from empirical data collected for the components, are used to build the decision tables. If a better component is available to use, then the currently executing (wrapped) component is monitored. When the component is in a safe state to swap out, then the new component(s) are initialized, the component location is updated, and the old component is removed.

4 Case Study

Initial experiments with the *KAROO* Framework have been conducted with excellent results. A data persistency service application and the associated MATLAB scripts were executed over an extended period of time, 8.601 hours, in order to collect a large sample (order of 10^3) of predicted violations and their outcomes (i.e., if the predicted violation is avoided by the adaptation of the system). The system has two functionally equivalent components for data security (level of encryption and key size used) and two for persistency (frequency of remotely saving information to a non-volatile memory). Also a threshold of 10% memory usage has been established; this threshold is quite small, in comparison to more conservative thresholds, e.g., 20%, that may be used in practice. The selection of appropriate components is done based on the status of the system and the relative importance of selected features.

The use of *KAROO* resulted in a total of 1812 swaps to avoid violating the constraint; only 37 were not executed in time, i.e., the memory threshold was crossed in 2.04% of the swaps. This would hypothetically crash the data persistency service application, but for experimental purposes, only the number of failed swaps was recorded. Though the threshold was exceeded 37 times, it should be noticed that the amount of memory used was on average only 0.36% above the 10% threshold. The detected failures are due not to wrong predictions but in almost all the cases to late reaction time. That is, *KAROO* predicted the constraint violation before it happened, but due to the overhead in the prediction, the overhead in the communication between MATLAB and C, and the scheduling of threads, the swap was not done in time and the used memory exceeded the threshold. The scheduling problem and the communication overhead can be minimized by the use of inter-process communication. A partial solution to decrease the overhead is the decomposition and conversion of parts of the scripts. Another alternative to decrease the number of failures is to increase the number of steps ahead used in the prediction. This would give more reaction time to the data persistency service application. On the other hand, the accuracy of the predictions decreases as the number of steps ahead increases, and a better synchronization algorithm is needed to avoid premature swaps. Another important aspect that should be noticed is that only part of the data is used to make the prediction. The more data that are available the more accurate the predictions. However, the overhead also increases with more data. In the experiment described here, a balance between accuracy and overhead was maintained by using only the last 200 data points to make a *five steps ahead* prediction. Alternative values of 100 and 300 data points were also investigated. Using 200 data points produced very accurate results with a small number of violations that could not be avoided due

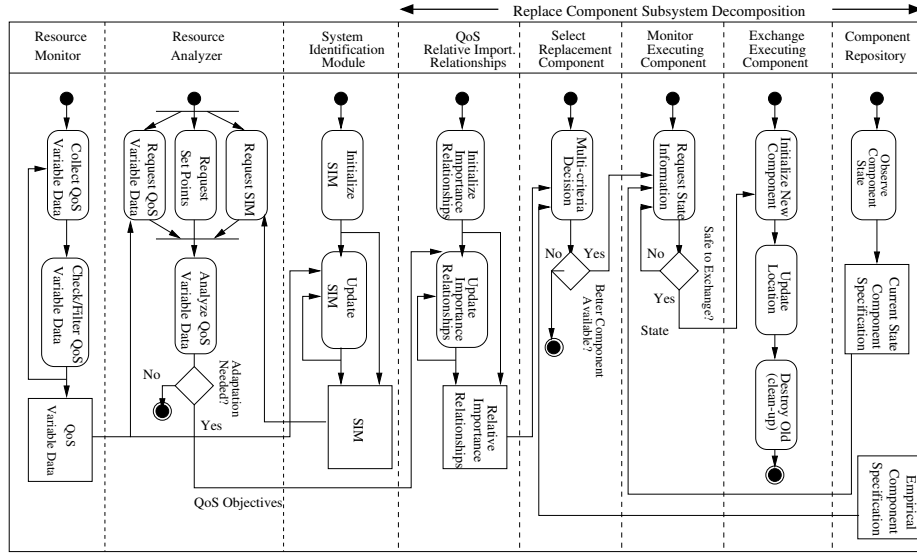


Figure 6. Dynamic View of the Architecture

to the overhead. In general, we conclude that this initial experiment with the *KAROO* approach has produced encouraging results showing the applicability and feasibility of the approach. We are aware that more validation is needed, and a validation plan is the subject of future work.

5 Conclusion

KAROO is a new technique to handle the dynamism that is rapidly becoming part of many companies' daily concerns. *KAROO* is powerful enough to handle applications running in a pre-specified but very constrained host such as most of the environments of embedded systems, and yet it is general enough to handle applications on heterogeneous platforms such as many web systems. For example, *KAROO* can be used in RNC (Radio Network Control) design to support monitoring and controlling of use of system resources (e.g., bandwidth, memory, and CPU utilization) in order to, for example, maximize revenue. *KAROO* may be used to adapt the system for variations in the demand for specific QoS requests and for variations in the characteristics of the traffic (e.g., burstiness) [20]. Also, *KAROO* has three major aspects that distinguishes it from other approaches.

Flexible Usage. *KAROO* is applicable in a wide variety of application domains for two main reasons. The first is that the framework largely decouples an application's domain specific capabilities from the adaptation capabilities. The domain specific capabilities are designed using component-based engineering techniques; the components used can be exchanged at run-time for other components that provide the same functional capabilities but have different QoS properties. The second reason is one of the

techniques used in the adaptation capabilities. A System Identification Module (SIM) is used that can dynamically identify the relationships between the inputs and the system resources. Rather than needing to know these relationships a-priori for a specific application, the *KAROO* Framework dynamically identifies these relationships and therefore allows it to be applied to a variety of different domains.

Predictable Adaptation. *KAROO* predicts when a non-functional or quality of service property will be violated, either from a resource constraint violation or a change in an environmental resource (e.g., a change in the security level). Prediction is achieved through the use of system identification techniques from control theory to capture the dominant behavior of a variable of interest (for example, memory usage). In the light of data collected by sensors, a non-parametric model is created and behavior is extrapolated to make the predictions. When a prediction is made, the system preparation to swap to a new component can be done in advance, resulting in an overall increase in the system's performance and robustness. Our experimental evaluation reveals the *KAROO* successfully adapted the system almost 98% of the time.

Automatic Reasoning. When a violation is predicted, *KAROO* dynamically exchanges components in the application to avert the violation using automated reasoning techniques that consider multiple, often conflicting, QoS criteria. A multi-criteria decision making (MCDM) approach is used to accomplish this.

References

- [1] Tarek F. Abdelzaher, K. Shin, and Nina Bhatti. Performance guarantees for web-server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, June 2001.
- [2] M. Beer and J. Whatley. A multi-agent architecture to support synchronous collaborative learning in an international environment. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, 2002.
- [3] A. Brown and K. Wallnau. Engineering of component-based systems. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*, pages 414–422, 1996.
- [4] Meunier R. Rohnert H. Sommerlad P. Bushmann, F. and M. Stal. *Pattern-oriented Software Architecture A System of Patterns*. John Wiley & Sons, 2001.
- [5] Joao W. Cangussu, Kendra Cooper, and Eric Wong. Multi criteria selection of components using the analytical hierarchy process. In *9th International SIGSOFT Symposium on Component-based Software Engineering (CBSE)*, Sweden, June 29th - 1st July 2006.
- [6] W. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 21)*, pages 635–643, Phoenix, AZ.
- [7] J. Y. Chung, J. W. Liu, and K. J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. Computers*, 39:1156–1174, September 1990.
- [8] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004.
- [9] Akos Ledeczki, Gabor Karsai, and Ted Bapty. Synthesis of self-adaptive software. In *Proceeding of the IEEE Aerospace Conference*, March 2000.
- [10] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Trans. Computers*, pages 58–68, May 1991.
- [11] Lennart Ljung. *System Identification: Theory for the user*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
- [12] Ying Lu, Tarek Abdelzaher, Chenyang Lu, and Gang Tao. An adaptive control framework for QoS guarantees and its application to differential caching services. In *10th IEEE International Workshop on Quality of Service*, pages 23–32, 2002.
- [13] David G. Luenberger. *Introduction to Dynamic Systems: Theory, models and applications*. John Wiley & Sons, New York, 1979.
- [14] R.T. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University, School of Computer Science, 1998.
- [15] Claudia Raibulet, Francesca Arcelli, Stefano Mussino, Mario Riva, Francesco Tisato, and Luigi Ubezio. Components in an adaptive and qos-based architecture. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 65–71, New York, NY, USA, 2006. ACM Press.
- [16] T. Saaty. *Fundamentals of Decision Making and Priority Theory*. RWS Publications, 1994.
- [17] Mary Shaw and David Garlan. *Software Architecture Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.
- [18] David C. Steere, Ashwin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158, 1999.
- [19] T. Uchiya, T. Katoh, T. Sukanuma, T. Kinoshita, and N. Shiratori. An architecture of agent repository for adaptive multiagent system. pages 802–813, 2002. Lecture Notes on Computer Science Vol.2344.
- [20] H. Vipat, P. Mathew, M. Castelino, and A. Tripathy. Network processor building blocks for all-ip wireless networks. *Intel Technology Journal*, 6(3), August 2002.
- [21] C. Wei, P. Hu, and O. Sheng. A knowledge-based system for patient image pre-fetching in heterogeneous database environments - modeling, design, and evaluation. *IEEE Transactions on Information Technology in Biomedicine*, 2001.
- [22] W. Zhang, S. Chen, L. Zhang, S. Yu, and F. Ma. Agarm: An adaptive grid application and resource monitor framework. pages 544–551, 2005. Lecture Notes on Computer Science Vol.3514.