Applying Adaptation Spaces to Support Quality of Service and Survivability[1]

Shawn Bowers
Oregon Graduate Institute
shawn@cse.ogi.edu

Lois Delcambre
Oregon Graduate Institute
lmd@cse.ogi.edu

David Maier
Oregon Graduate Institute
maier@cse.ogi.edu

Crispin Cowan
Oregon Graduate Institute
crispin@cse.ogi.edu

Perry Wagle
Oregon Graduate Institute
wagle@cse.ogi.edu

Dylan McNamee
Oregon Graduate Institute
dylan@cse.ogi.edu

Anne-Françoise Le Meur
IRISA/INRIA Rennes, Campus
Universitaire de Beaulieu
lemeur@irisa.fr

Heather Hinton
Ryerson Polytechnic Institute
heather@eecg.utoronto.ca

Abstract

Adaptation is a key technique in constructing survivable information systems. Allowing a system to continue running, albeit with reduced functionality or performance, in the face of reduced resources, attacks, or broken components is often preferable to either complete shutdown or continued normal operation in compromised mode. However, unpredictable adaptation can sometimes be worse than the problem it seeks to cope with. In this paper we introduce adaptation spaces, which precisely and predictably specify the adaptation of a software component. We then present two survivable systems that have been specified and implemented using adaptation spaces. The first example uses user preferences regarding quality in an audio application to guide the adaptation when available bandwidth decreases. The second trades off performance overhead with intrusion resistance for "stack-smashing" attacks. We formally define an adaptation space and show briefly how it enables certain kinds of reasoning about adaptive applications. We conclude with related work and future plans.

1. Introduction

The need for dynamically adapting software is rising due to increases in computing power, software complexity, and networked systems. Incorporating adaptive solutions into software introduces flexibility and robustness. However, it also introduces a challenge for software designers – specifically, how to correctly specify the adaptation requirements of the system. A major problem with automatic adaptation is the possibility that the response will make the situation worse.

We propose a solution to the problem of correctly specifying adaptation requirements by introducing *adaptation spaces*. An adaptation space is a formalism that allows us to specify when various implementation alternatives of a software component are feasible and when they should be selected, based on current conditions of the application environment and the preferences of the user. Each implementation alternative meets the basic goals for the adaptable software component, but the implementation alternatives differ in the level of service that they provide. The forms of implementation alternatives vary. They might be represented as a program written in a programming language, a query plan, or a collection of parameter settings. They also differ in their feasibility, that is, in the conditions under which they can run.

We have demonstrated the use of adaptation spaces in two applications: the Adaptive Audio Quality application and the StackGuard Adaptation Manager (SAM). In the Adaptive Audio Quality application, user preference, i.e., the user's quality of service specification, guides the choice of an implementation alternative. In SAM, the security strategy of a distributed system is guided by the level of paranoia, which is based on the number of intrusion attempts on the system.

We characterize an adaptive component of an application as a component with more than one implementation alternative. The choice of which

[1] This work is supported by the Defense Advanced Research Projects Agency, DARPA order number E299, monitored by the US Army Research Laboratory under grant F30602-96-1-0302 (Heterodyne) and DARPA order number E300, monitored by the US Army Research Laboratory under grant F30602-96-1-0331 (Immunix).

implementation alternative to use is based on its feasibility in the current situation and the user's preference. For example, in the Adaptive Audio Quality application, the implementation alternatives differ in their sample size (i.e., number of bits per sample), sample frequency (i.e., number of samples per second), and number of channels (e.g., stereo or mono). The feasibility of each implementation alternative is determined by comparing the available bandwidth with the bandwidth required for the implementation alternative. Preference information allows the user to state, for example, that he or she prefers sample frequency to the other aspects of the audio transmission. The preference information is used to select an implementation alternative when there are multiple, feasible implementation alternatives.

Both SAM and a prototype implementation of the Adaptive Audio Quality application used the adaptation space *navigator*. The adaptation space navigator is a tool that takes as input an adaptation space along with the application's current state and selects the appropriate implementation alternative. These examples show that an adaptation space can successfully specify the feasibility and preference information for two diverse applications, at two, dramatically different choices of granularity. In this paper we discuss the adaptation space formalism and we show how these two diverse applications can be correctly and compactly described in the formalism.

We introduce adaptation spaces in Section 2 and then highlight the two adaptive applications that we have implemented using adaptation spaces in Sections 3 and 4. Both implementations use the adaptation space navigator, developed as part of this work, to select an implementation alternative when the conditions change. The adaptation space formalism is briefly presented in Section 5, with the specification of the two applications. Related work is briefly reviewed in Section 6. The paper concludes with a discussion of work in progress and plans for future work in Section 7.

2. Introduction to Adaptation Spaces

An adaptation space describes a single software component that has multiple implementation alternatives. An implementation alternative might correspond to any kind of reconfiguration, ranging from changing a single bit of state to a complete replacement of the implementing software. To describe the feasibility of the implementation alternatives, we must identify the conditions of interest to this component. That is, we must identify the external conditions that will prompt the choice of a different implementation alternative. Note that the mechanism for monitoring the external conditions as well as the mechanism used to switch from one implementation alternative to another is not specified by the adaptation space itself.

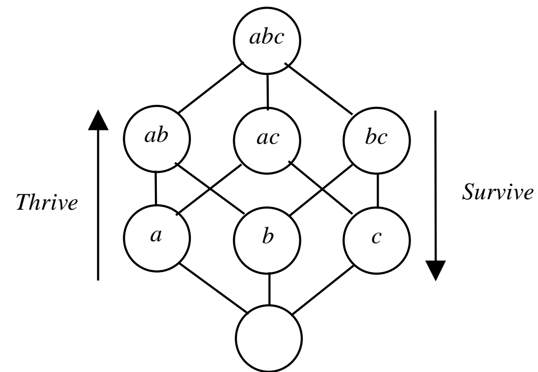


Figure 2.1. Simple condition space.

An adaptation space includes a specification of the condition space, the cross-product of the conditions of interest (i.e., conditions that might induce a change to a different version of the component). Figure 2.1 shows a simple condition space with three Boolean conditions *a*, *b*, and *c*. The top node represents the case where *a*, *b*, and *c* hold, while the bottom node represents the case where none of the conditions hold. Downward adaptations switch to an implementation that can function correctly with fewer assumptions about the execution environment (i.e., fewer true conditions), and are said to be *surviving adaptations*. Upward adaptations switch to implementations that require more assumptions about the execution environment, but also provide additional functionality or quality of service, and are said to be *thriving adaptations*. Note this introduces a convention. Since for a condition *a*, we could swap the meanings of *a* and $\neg a$, and still have a Boolean expression, we have a convention that *a* be more restrictive than $\neg a$ (e.g., subsystem available, more bandwidth available).

The size of the condition space is exponential in the number of independent Boolean conditions. Thus the check to make sure that an adaptive system responds properly to all conditions is exponential, at face value.[2] Note that if the conditions of interest for an application are not independent, then the condition space can be simplified based on the logical dependence among conditions.

For each implementation alternative, we must specify when it is feasible. That is we must specify under what conditions it is feasible to run this implementation alternative. We define a *slice* of a condition space as the subset of nodes in the condition space where a particular implementation alternative is feasible. In our current

[2] This difficulty is general to adaptive systems, and not unique to adaptation spaces.

work, we specify a slice through a *minimum* condition. For example, the slice on the left side of Figure 2.2 is characterized by conditions *a* and *b* being true. That is, all nodes in the slice have both *a* and *b* true. Similarly, the slice on the right side of Figure 2.2 is characterized by condition *c* being true.

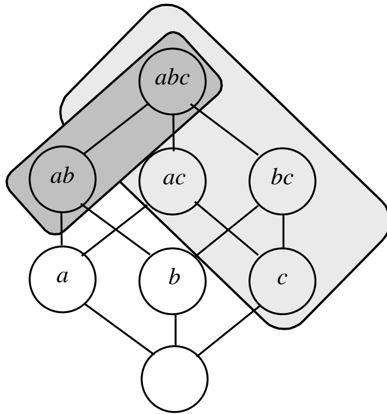


Figure 2.2. Slices of a condition space.

The adaptation space formalism also allows for the specification of preference. Preference places an ordering on implementation alternatives. Whenever there is more than one feasible implementation alternative, the preference configuration is used to break the tie. Preference can change at run-time. The system designer can use information about the application, the current environment, the implementation alternatives, etc., to help set the preference configuration.

The *adapt operator*, defined as part of the adaptation space formalism, selects the appropriate implementation alternative to use for the current conditions and current preference configuration. The adapt operator guarantees that an implementation alternative will never be selected when it is infeasible and that an implementation alternative will be selected according to the preference ordering.

We have applied the adaptation space formalism to the problem of describing the Adaptive Audio Quality application and the Security Adaptation Manager (SAM) application. We developed and implemented the adaptation space navigator, which was successfully used by both SAM and with a prototype implementation of the Adaptive Audio Quality application. The adaptation space navigator implements the adapt operator and uses the eXtensible Markup Language (XML) [1] format for adaptation space specification.

The Adaptive Audio Quality application and SAM deal with different aspects of survivable information systems. In both cases, we are looking at a predictable, adaptive trade off in system functioning, instead of erratic behavior or complete failure. In the case of audio adaptation, we want to trade off resource availability (network bandwidth) against delivered Quality of Service (QoS). Rather than randomly lose pieces of a full audio stream, audio quality is scaled back to match available bandwidth, according to user preferences. For SAM, we are adapting perceived threat of intrusion, by trading system performance against the degree of intrusion resistance. An important point in both cases is the ability of a component to thrive as well as survive. That is, in addition to the ability to trade off system functionality or performance when resources are lacking or the system is under attack, we want to restore functionality and performance when conditions improve (see Figure 2.1).

3. Adaptation Spaces for Adaptive Multimedia QoS Management

One challenge in implementing multimedia systems is deciding what to do whenever the presentation requires more resources than are available [2]. This happens frequently in client-server multimedia systems because the available bandwidth changes as the application is running. One way to address this challenge is to perform adaptive quality of service (QoS) management [2]. When loss of resources occur, instead of halting a presentation altogether, or letting quality deteriorate abruptly, adaptive QoS management degrades the quality of the presentation gracefully (e.g. by reducing the resolution of the transmitted video stream) in response to the change in resources.

There is often more than one way to reduce the quality of a presentation. In video presentations, both resolution and frame rate can be lowered in order to adapt to decreasing bandwidth. The decision on how the presentation should be adapted is based on the preference of the user. For example, the user of a video presentation might consider the video resolution to be more important than the frame rate. Then the frame rate of the video presentation will be reduced before the resolution is reduced when the available bandwidth drops below the bandwidth required for a normal presentation. The user can also specify the tolerable quality levels. That is, a user can decide that the video presentation is not worth showing below a certain resolution.

3.1. The Adaptive Audio Quality Application

The Adaptive Audio Quality framework is intended to support adaptive QoS management of audio signals sent over a network [3]. Figure 3.1 depicts a high level view of one possible application of the Adaptive Audio Quality framework, which supports adaptive QoS management in a client-server audio player. The server sends audio streams over a network to a client, which plays the audio

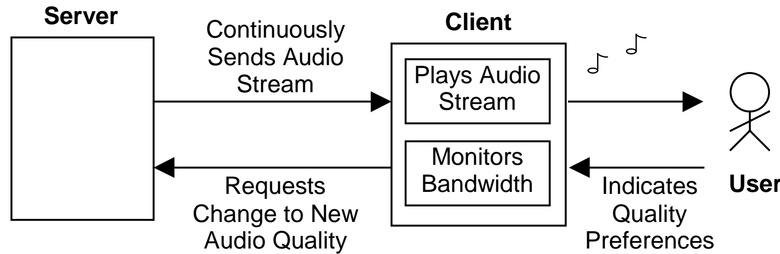


Figure 3.1. The Adaptive Audio Quality application.

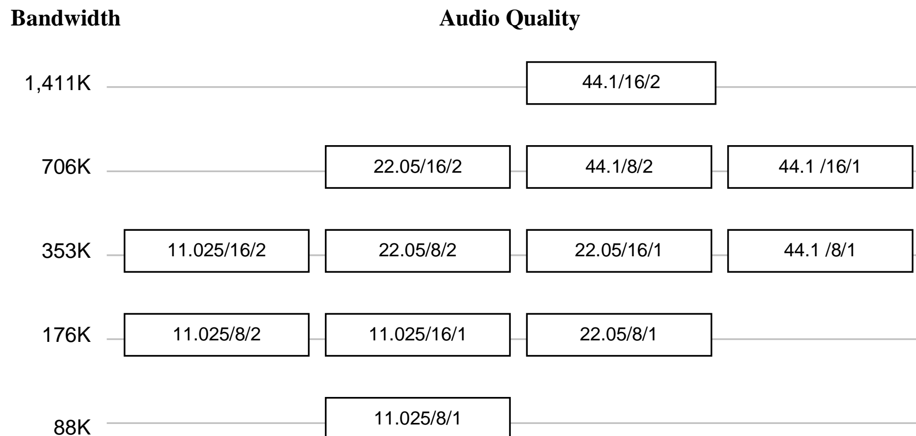


Figure 3.2. The implemented audio qualities with their bandwidth requirements.

stream and continually monitors bandwidth. If the monitored bandwidth drops below the current requirement, the client sends a message to the server to request lower audio quality. If the bandwidth increases enough to support a higher quality audio stream, the client sends a request to the server asking for higher audio quality.

Three attributes are used to specify audio quality: frequency, sample size, and the number of channels. Frequency is the number of samples per second, sample size is the number of bits per sample, and the number of channels is either mono or stereo (i.e., 1 or 2, respectively). Note that all three attributes require more bandwidth as quality is improved.

Figure 3.2 shows the combinations of values for the three quality attributes that are implemented for this application. For instance, the node in Figure 3.2 labeled "44.1/16/2" represents an audio transmission with frequency of 44.1 K samples per second (KS/s), 16 bits per sample, and 2 channels (i.e., stereo). The required bandwidth value for each of the audio qualities is shown on the left side of Figure 3.2. As an example, a bandwidth of at least 1,411 K bits per second (Kb/s) is required for the audio quality labeled "44.1/16/2" and a bandwidth of at least 176 Kb/s is required for the "11.025/8/2", "11.025/16/1", as well as the "22.05/8/1" quality levels. Thus, if the bandwidth is 176 Kb/s, for example, we can choose one of these three implementations.

The user indicates the quality attributes (e.g., frequency) and values (e.g., 44.1 KS/s) that they prefer by assigning them weights as the application is running. As an example, the weighting assignments in Figure 3.3 results in a preference configuration for the Adaptive Audio Quality application. This preference configuration reflects a user who cares most about the frequency and then about the sample size. Thus, frequency is weighted higher than sample size, and the sample size is weighted higher than the number of channels.

Quality Attrbiute	*Weight*
Frequency (Wf)	**0.8**
Sample Size (Ws)	**0.3**
Channels (Wc)	**0.1**

Frequency Value	*Coefficient*
44.1K (Wf_{44})	**0.8**
22.05K (Wf_{22})	**0.4**
11.025K (Wf_{11})	**0.0**

Sample Size	*Coefficient*
16bps (Ws_{16})	**0.8**
8bps (Ws_{8})	**0.2**

Channel Value	*Coefficient*
2 – stereo (Wc_{2})	**0.6**
1 – mono (Wc_{1})	**0.4**

Figure 3.3. A possible weight and coefficient assignment.

Each value for a quality attribute is also weighted by *coefficients,* as shown in Figure 3.3. The formula to calculate audio quality preference is shown in Figure 3.4.

Quality Attribute Preference Formula

Assuming the requested frequency is *x*, sample size is *y*, and number of channels is *z*, its preference can be computed as:

$$\mathbf{pref = Wf*Wf_x + Ws*Ws_y + Wc*Wc_z}$$

Figure 3.4. The formula to calculate audio quality preference.

Figure 3.5 shows the audio quality path and audio quality preferences that result from applying the formula of Figure 3.4 to the weightings of Figure 3.3. By assigning weights to the quality attributes and coefficients to the values, any of twelve possible paths through Figure 3.2 can be specified [3].

As shown in Figure 3.5, the audio quality with the highest preference in the preference configuration for the current bandwidth range is selected whenever the bandwidth enters a new range. The client sends the request for this new audio quality to the server. The server configures appropriate filters to adjust the quality of the audio signal before sending it to the client. Each filter affects a particular quality dimension (i.e. frequency, sample size, or number of channels). Figure 3.6 shows a revised view of the Adaptive Audio Quality application that includes the server's role of constructing the pipeline.

3.2. The Adaptive Audio Quality Adaptation Space

The choice of implementation alternatives for the Adaptive Audio Quality application based on the available bandwidth and the current preferences of the user has been specified and implemented as an adaptation space. Each implemented audio quality combination shown in Figure 3.2 is modeled as an implementation alternative and the feasibility of each implementation alternative is based on the amount of bandwidth available. For example, in order for the audio quality "44.1/16/2" implementation alternative to be feasible, the condition that bandwidth be greater than or equal to 1,411 Kb/s must be true. In Figure 3.5 we see the feasibility information in the form of the minimum bandwidth requirements listed to the left of the implementation alternatives. The implementation alternatives that appear to the right of a bandwidth (in the same row) are feasible whenever the available bandwidth meets or exceeds the minimum bandwidth condition. For example, the slice defined by the condition bandwidth ≥ 706 Kb/s has three implementation alternatives associated with it: "22.05/16/2", "44.1/8/2," and "44.1 /16/1."

In the Adaptive Audio Quality application, coefficients and weights are used to specify preference. The preferred implementation alternative for each slice corresponds to the implementation alternative with the highest preference of the alternatives from each slice, as shown by the shading in Figure 3.5. As an example, for the slice defined by the condition that bandwidth is greater than or equal to 176 Kb/s, the implementation alternative "22.05/8/1" is the preferred one since its preference of .42 is higher than the preferences of "11.025/8/2" and "11.025/16/1".

An adaptation space must specify when a given implementation alternative can directly replace another. That is, an adaptation space specifies a transition graph where the implementation alternatives are the nodes. In the Adaptive Audio Quality application the transition graph in the adaptation space application is a complete graph. There is a transition between every pair of

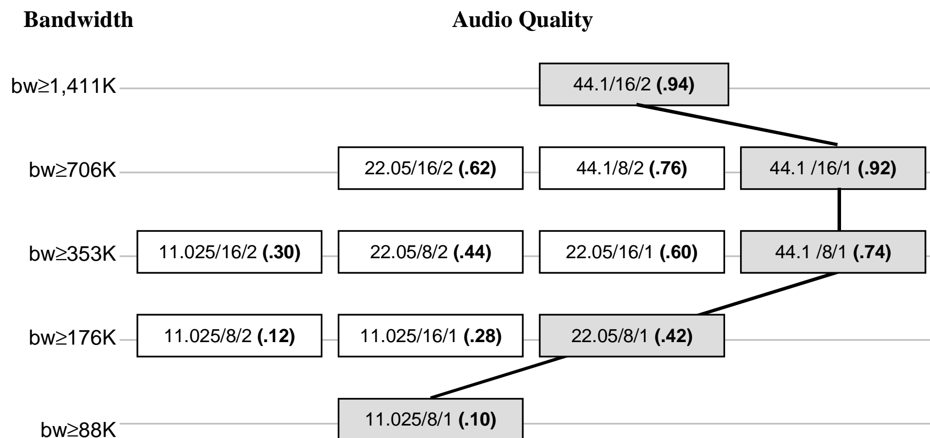


Figure 3.5. Path and rankings from applying the formula of Figure 2.4 to the weights and coefficients of Figure 3.3.

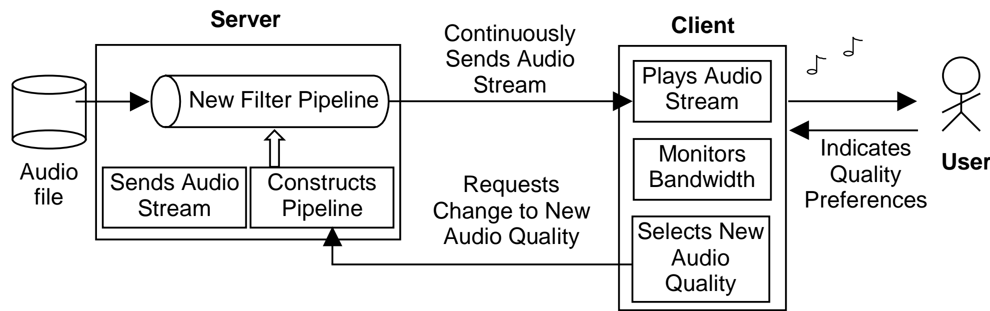


Figure 3.6. Revised view of the Adaptive Audio Quality application.

implementation alternatives in the application. When a transition graph is not complete, the choice of an implementation alternative is limited to the implementation alternatives that are connected to the current implementation in the transition graph.

4. Adaptation Spaces for Survivable Systems

Commercial software often contains bugs that can lead to security vulnerabilities [4]. We are working on ways to enhance existing software so that it can survive security attacks. Such a software system, called a *survivable system*, can continue performing its necessary tasks in the event of successful security attacks [5].

The StackGuard compiler [6] is a compiler enhancement that protects existing programs against stack-smashing attacks. Stack-smashing attacks exploit buffer overflow vulnerabilities, which are the most commonly reported software security problem. A buffer overflow vulnerability occurs when there is a lack of bounds checking performed on the size of input being stored in an array. Attackers exploit buffer overflow vulnerabilities by purposely overflowing the buffer with executable attack code, including a new return address that points to the new code. This scenario is illustrated in Figure 4.1a. Note that a full array bounds checking compiler would be secure against buffer overflow attacks, however, such compilers are not yet available for commercial use [7].

Stackguard inserts a *canary*[3] value before the function return address on the stack. The canary is checked prior to jumping to the address pointed to by the return address. If the canary has been overwritten, e.g., by the attack code, a message is reported and the program is halted. Figure 4.1b illustrates the canary inserted by StackGuard.

The protection that is provided by the StackGuard compiler is not impenetrable. A clever attacker can succeed in attacking by leaving the canary unchanged [7].

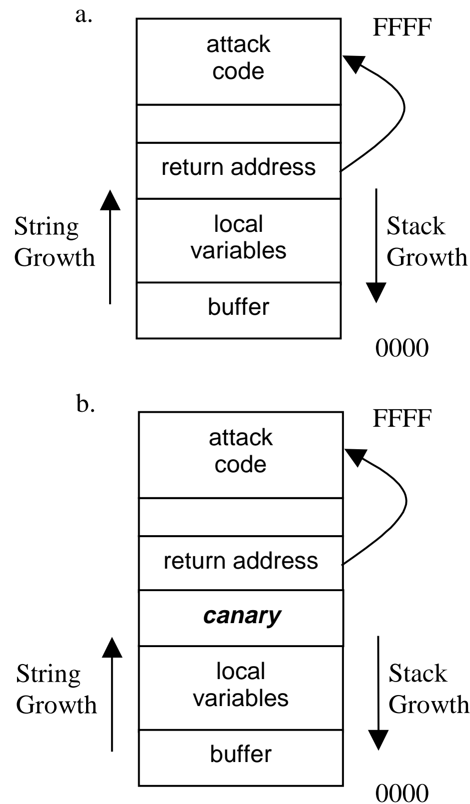


Figure 4.1. The StackGuard detection mechanism: (a) result of a stack-smashing attack and (b) StackGuard's inserted canary for detection.

The StackGuard compiler can provide three levels of security that correspond to the type of canary used. The three types of canaries are called *terminator*, *terminator with diversity*, and *random*. A *terminator* canary uses a string termination symbol as the canary value. This makes it difficult for the attacker to deposit a termination string symbol in the canary's location as well as alter the return value above the canary symbol (see Figure 4.1b) [7]. The *terminator with diversity* canary places a termination

[3] The canary value is analogous to the canaries used in mining to detect a lack of oxygen.

string symbol into the canary value but also adds a random number of values between the canary and return address. This makes it difficult for the attacker to predict the location of the canary. Finally, a *random* canary requires the attacker to guess the canary value, a random number. The random canary is the most secure and also the slowest of the three approaches. The terminator with diversity canary is less secure but faster than the random canary approach. The terminator canary is the fastest of the three approaches and the least secure.

4.1. The Security Adaptation Manager

The Security Adaptation Manager (SAM) tool monitors stack-smashing attacks attempted on the programs of a software system [7]. Each attack is classified according to the privileges of the program being attacked (*root*, *user*, and *nobody)* and the sensitivity of the host on which the program is running (*security-server*, *other-server*, and *workstation).*

Each program in the software system monitored by SAM is compiled using the three different levels of StackGuard protection. Based on the number of attacks being made as well as the type of programs and hosts being attacked, SAM adapts to the appropriate StackGuard version for each class of programs by using a security deployment. There are nine classes of programs, where each program class corresponds to a particular type of program privilege (i.e., root, user, or nobody) combined with a particular type of host sensitivity (i.e., security, other, or workstation). Figure 4.2 shows one possible security deployment.

	Security	**Other**	**Workstation**
Root	Random	Terminator	Terminator
User	Terminator	Terminator	Terminator
Nobody	Terminator	Terminator	Terminator

Figure 4.2. A security deployment.

In the security deployment shown in Figure 4.2, all programs running on the security server with root privileges use a random canary. All other classes of programs have been compiled using a terminator canary.

SAM uses a similar matrix to define conditions where a condition portrays the current level of paranoia. These three conditions are named *calm*, *nervous*, and *panicked* and are shown in Figure 4.3. Each condition sets a minimum threshold for the number of attacks experienced in a specific time frame against programs at the root, user, and nobody levels running on security, other, and workstation servers.

Calm

	Security	**Other**	**Workstation**
Root	≤ 1	≤ 5	≤ 15
User	≤ 5	≤ 10	≤ 20
Nobody	≤ 5	≤ 15	≤ 25

Nervous

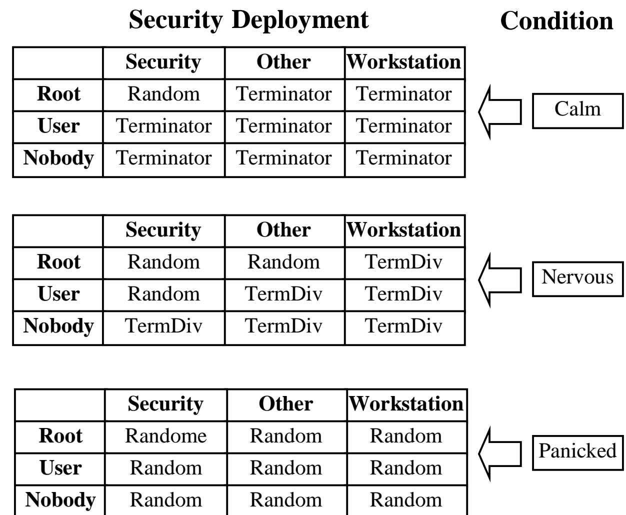
	Security	**Other**	**Workstation**
Root	≤ 2	≤ 10	≤ 20
User	≤ 10	≤ 15	≤ 30
Nobody	≤ 10	≤ 20	≤ 40

Panicked

	Security	**Other**	**Workstation**
Root	≤ 3	≤ 15	≤ 25
User	≤ 15	≤ 20	≤ 40
Nobody	≤ 15	≤ 25	≤ 50

Figure 4.3 The conditions used by SAM.

The adaptation mechanism in SAM must select a security deployment, given the current condition (i.e., the current state). Figure 4.4 shows one possible condition assignment (where terminator with diversity is denoted *TermDiv*).



Security Deployment — **Condition**

	Security	**Other**	**Workstation**
Root	Random	Terminator	Terminator
User	Terminator	Terminator	Terminator
Nobody	Terminator	Terminator	Terminator

Calm

	Security	**Other**	**Workstation**
Root	Random	Random	TermDiv
User	Random	TermDiv	TermDiv
Nobody	TermDiv	TermDiv	TermDiv

Nervous

	Security	**Other**	**Workstation**
Root	Randome	Random	Random
User	Random	Random	Random
Nobody	Random	Random	Random

Panicked

Figure 4.4. Security deployments for each condition.

4.2. The SAM Adaptation Space

In the SAM adaptation space, the current condition is represented by the current state of the system, which is calm, nervous, or panicked. Accordingly, the condition space has only three nodes.

The implementation alternative choices in the SAM adaptation space correspond to the security deployments of Figure 4.4. The feasibility of each alternative is defined by a single condition. Although currently there is only one implementation alternative associated with each slice of the condition space, it is not hard to imagine multiple security deployments for a given condition. In such a case, a method of determining preference between these additional implementation alternatives (e.g., using weights and coefficients) would need to be determined by SAM.

For the SAM adaptation space, the slice defined by the calm condition is the most preferred, the slice associated with the nervous condition is the next most preferred, and the slice associated with the panicked condition is the least preferred. This slice ordering corresponds to SAM's preference of implementation alternatives. When the calm condition holds, all three security deployments are feasible (i.e., calm implies nervous, which implies panicked). In this case, the slice ordering specifies that the implementation associated with the calm condition is the most preferred. A similar case exists when the nervous condition is true.

Finally, any security deployment can be switched directly with any other, which means that the transition graph for the SAM adaptation space is fully connected.

5. The Adaptation Space Model

5.1. The Formalism

An adaptation space is defined as a quintuple, which consists of a set of conditions, a set of implementation alternatives, a set of slices, a transition graph, and a set of slice-orderings. Table 1 presents the formal definitions of the adaptation space model.

In defining an adaptation space, the condition set C represents the conditions of interest to the application. Each condition is a Boolean variable. The set of implementation alternatives I represents the set of implemented versions, or configurations, of the adaptable component. The set of slices S specifies the feasibility information for each of the implementation alternatives of the application.

A single slice defines the feasibility of a particular set of implementation alternatives as a subset of the conditions that can occur for the application. As shown in Table 1, a slice is defined by a subset of conditions and is associated with zero or more implementation alternatives, precisely the implementation alternatives that use this slice to define their feasibility.

The transition graph T consists of a set of available transitions, i.e., the available replacements of one implementation alternative directly for another. In our two examples, T was always a complete graph, but in general it may not be possible to move from a given implementation to all the alternative implementations.

The set of slice-orderings Σ consists of all the slice orderings that are meaningful for the application. Note that some applications have only one slice ordering of interest. A slice ordering σ places a total order on the slices in the adaptation space to represent the user's preference for slices, where higher preference slices are ahead of lower preference slices. Note that Σ does not completely determine preference, in general. It may contain more than one ordering, and a given slice in an

Table 1. The Adaptation Space Model

Symbol	Representation	Definition
A	Adaptation space	$A = (C, I, S, T, \Sigma)$
C	Set of conditions	$C = \{c_1, c_2, \ldots, c_l\}$ where each $c \in C$ represents a Boolean condition.
I	Set of implementation alternatives	$I = \{i_1, i_2, \ldots, i_m\}$ where each $i \in I$ represents an implementation alternative.
S	Set of slices	$S = \{s_1, s_2, \ldots, s_n\}$ where each $s_i \in S$ is a pair (C_i, I_i) such that $C_i \subseteq C$ and $I_i \subseteq I$. For any two slices $s_i, s_j \in S$ where $s_i = (C_i, I_i)$ and $s_j = (C_j, I_j)$, if $s_i \neq s_j$ then $C_i \neq C_j$. The set $\{I_1, I_2, \ldots, I_n\}$ in S partitions I.
T	Transition graph	$T = \{t_1, t_2, \ldots, t_p\}$ where each $t \in T$ is a pair (i_i, i_j) such that $i_i, i_j \in I$, $i_i \neq i_j$. Each pair (i_i, i_j) represents a transition from i_i to i_j.
Σ	Set of slice-orderings	$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_q\}$
σ	Slice-ordering	$\sigma = (s_1, s_2, \ldots, s_n)$ where $s_1, s_2, \ldots, s_n \in S$, n is the number of slices in S, each s in σ is unique, and the slices in σ are ordered according to preference with s_1 the most preferred and s_n the least preferred.

We require that $|S| \geq 1$, $|I| \geq 1$, and $|\Sigma| \geq 1$. Non-trivial adaptation spaces have $|I| \geq 2$, $|C| \geq 1$, and $1 \leq |T| \leq n*(n-1)$.

Table 2. Adaptation Space Functions

Function	Description	Definition
δ	The defining condition set function	$\delta: i \rightarrow C'$, where $i \in I$, $C' \subseteq C$. For every i there exists exactly one $s \in S$, $s = (C', I')$ and $i \in I'$. $\delta(i) = C'$.
ϕ	The implementation alternative currently preferred in each slice	$\phi: s \rightarrow i$, where $s = (C', I') \in S$ and $i \in I'$. $\phi(s) = i$, where i is one particular element of I', chosen by the user.
ρ	The preference configuration function	$\rho: \sigma \times \phi \rightarrow \pi$, where for $\sigma = (s_1, s_2, \ldots, s_n) \in \Sigma$, $\rho(\sigma, \phi) = \pi = (\phi(s_1), \phi(s_2), \ldots \phi(s_n))$.
adapt	Computes the most preferred implementation alternative among the feasible implementation alternatives	$Adapt: C' \times \pi \times i \times A \rightarrow i'$, where $C' \subseteq C$, π is the current preference configuration, i is the current implementation alternative, $A = (C, I, S, T, \Sigma)$, and $i, i' \in I$.

ordering may hold more than one implementation alternative.

Table 2 defines the four functions used with adaptation spaces, which include the defining condition set function, the most preferred implementation function, the preference configuration function, and the adapt operator.

The defining condition set function δ is defined simply for convenience to give the minimum set of conditions that define feasibility for an implementation alternative. The most preferred implementation alternative function, ϕ, is used with the slice ordering to indicate preference. Function ϕ gives the most preferred implementation alternative for a given slice. The preference configuration function, ρ, combines the information contained in ϕ with the current slice ordering σ to produce a total order of the most preferred implementation alternatives.

The adapt operator finds the most preferred, feasible implementation alternative under the current setting of conditions specified in the adaptation space for the application. The adapt operator takes a set of currently true conditions, the current preference configuration, a current implementation (typically the last result of *adapt*), and an adaptation space to compute the most preferred implementation to transition to. The algorithm for the adapt operator is shown in Figure 5.1.

Table 3. The Adaptive Audio Quality Adaptation Space

Symbol	Representation	Value
A	Adaptation space	$A = (C, I, S, T, \Sigma)$
C	Set of conditions	$C = \{bw_{1411}, bw_{706}, bw_{353}, bw_{176}, bw_{88}\}$, where bw_X represents the condition "$bw \geq$ X".
I	Set of implementation alternatives	I = {"44.1/16/2", "44.1/16/1", "44.1/8/2", "44.1/8/1", "22.05/16/2", "22.05/16/1", "22.05/8/2", "22.05/8/1", "11.025/16/2", "11.025/16/1", "11.025/8/2", "11.025/8/1"}.
S	Set of slices	$S = \{s_{1411}, s_{706}, s_{353}, s_{176}, s_{88}\}$, where: s_{1411} = ({bw_{1411}}, {"44.1/16/2"}) s_{706} = ({bw_{706}}, {"44.1/16/1", "44.1/8/2", "22.05/16/2"}) s_{353} = ({bw_{353}}, {"11.025/16/2", "22.05/8/2", "22.05/16/1", "44.1/8/1"}) s_{176} = ({bw_{176}}, {"11.025/8/2", "11.025/16/1", "22.05/8/1"}) s_{88} = ({bw_{88}}, {"11.025/8/1"})
T	Transition graph	The transition graph is fully connected, which means that for every $i_1, i_2 \in I$, such that $i_1 \neq i_2$, there exists a $t = (i_1, i_2) \in T$.
Σ	Set of slice-orderings	$\Sigma = \{(s_{1411}, s_{1706}, s_{353}, s_{176}, s_{88})\}$.

adapt-OPERATOR

```
adapt(C', π, i, A)
   for each imp ∈ π in order from
            highest to lowest
      C = δ(imp)
      if C' implies C and there is a
            transition i to imp ∈ T
         return imp
   return error
```

Figure 5.1 The *adapt* operator.

5.2. The Adaptive Audio Quality and SAM Adaptation Spaces

We return here to the adaptation space examples of Sections 3 and 4, to show their descriptions in the formal notation. Table 3 formally defines the adaptation space for the Adaptive Audio Quality application presented in Section 3. The formal definition of the SAM adaptation space is shown in Table 4. Note that in each case we have only one slice ordering.

The syntax used in Tables 3 and 4 is only for purposes of exposition. The actual format we used in practice is defined in XML [1] so that we can make use of existing tools, such as XML parsers, for processing adaptation space specifications. The XML format is also capable of expressing certain kinds of preference, such as the weights and coefficients used in the audio example. In our implementation work, the adapt operator is realized through the adaptation space navigator, which is initialized from an adaptation space specification, and calculates the current implementation alternative based on conditions and preferences.

5.3. Analysis of the Adaptation Space Formalism

One of the main motivations for the adaptation space formalism was the formal definition of the adapt operator. The adapt operator gives an application the ability to adapt in response to changes in external conditions at runtime. The adaptation space formalism, together with

Table 4. The SAM Adaptation Space

Symbol	Representation	Value
A	Adaptation space	$A = (C, I, S, T, \Sigma)$
C	Set of conditions	$C = \{calm, nervous, panicked\}$, where u = user, r = root, n = nobody, s = security, o = other, and w = workstation in: $calm$ = r-s-attacks ≤ 1 ∧ r-o-attacks ≤ 5 ∧ r-w-attacks ≤ 15 ∧ u-s-attacks ≤ 5 ∧ u-o-attacks ≤ 10 ∧ u-w-attacks ≤ 20 ∧ n-s-attacks ≤ 5 ∧ n-o-attacks ≤ 15 ∧ n-w-attacks ≤ 25 $nervous$ = r-s-attacks ≤ 2 ∧ r-o-attacks ≤ 10 ∧ r-w-attacks ≤ 20 ∧ u-s-attacks ≤ 10 ∧ u-o-attacks ≤ 15 ∧ u-w-attacks ≤ 30 ∧ n-s-attacks ≤ 10 ∧ n-o-attacks ≤ 20 ∧ n-w-attacks ≤ 40 $panicked$ = r-s-attacks ≤ 3 ∧ r-o-attacks ≤ 15 ∧ r-w-attacks ≤ 25 ∧ u-s-attacks ≤ 15 ∧ u-o-attacks ≤ 20 ∧ u-w-attacks ≤ 40 ∧ n-s-attacks ≤ 15 ∧ n-o-attacks ≤ 25 ∧ n-w-attacks ≤ 50
I	Set of implementation alternatives	$I = \{calm_imp, nervous_imp, panicked_imp\}$
S	Set of slices	$S = \{s_1, s_2, s_3\}$, where: $s_1 = (\{calm\}, calm_imp)$, $s_2 = (\{nervous\}, \{nervous_imp\})$ $s_3 = (\{panicked\}, \{panicked_imp\})$
T	Transition graph	The transition graph is fully connected, which means that for every $i_1, i_2 \in I$, such that $i_1 \neq i_2$, there exists a $t = (i_1, i_2) \in T$.
Σ	Set of slice-orderings	$\Sigma = \{(s_1, s_2, s_3)\}$.

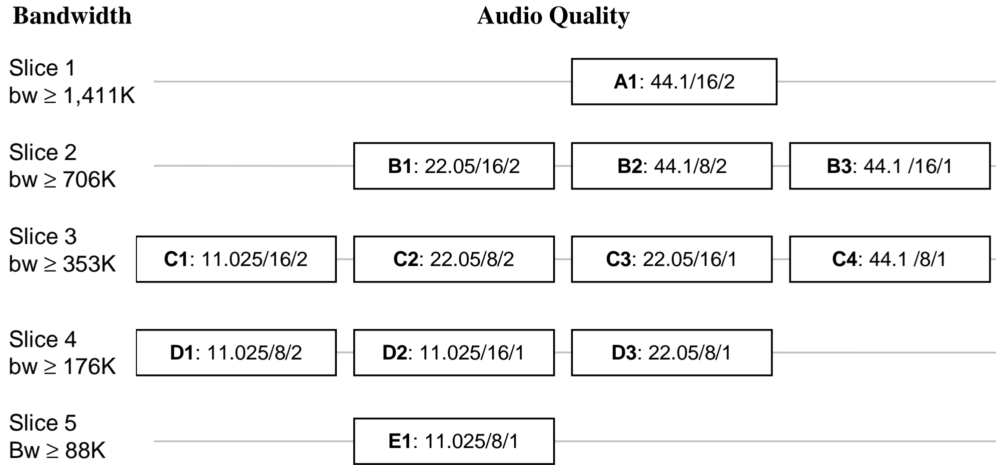


Figure 5.2. Implementation alternatives of the Adaptive Audio Quality application.

the algorithm for the adapt operator, guarantees that the runtime transitions are safe, feasible, and that they adhere to the current preference configuration. Another important property of the adapt operator is that it is linear in the number of slices, where the number of slices is less than or equal to the number of implementation alternatives.

The definition of a preference configuration specifies preference at two levels: first at the slice level, through the slice order σ, then within a slice, through the ϕ function. Constraining preference to follow a slice order does not limit the ability to express preferences. Consider the implementation alternatives for the adaptive audio quality example, shown in Figure 5.2. The implementation alternatives are labeled A1 through E1. Suppose a preference configuration of A1, B2, C2, B3, D1, E1 is desired. The interesting part of this preference configuration is B3; it violates our definition of ϕ as a function because the slice defined by the condition bandwidth ≥ 706 Kb/s has two implementation alternatives, B2 and B3. B3 will never be selected by the adapt operator. If bandwidth is ≥ 706 Kb/s, then B2 will be selected. There is no possibility of selecting B3, because B2 is preferred to B3. Thus, A1, B2, C2, B3, D1, E1 is equivalent to A1, B2, C2, D1, E1, which can be expressed in the formalism. B3 is an example of dead code, for this preference configuration (although it could be "resurrected" if user preferences change).

There are other situations where unused application alternatives can be determined a priori. Suppose that the slice order for the Adaptive Audio Quality application is (Slice 1, Slice 3, Slice 2, Slice 4, Slice 5) for the slices shown in Figure 5.2. The interesting part of this slice order is Slice 2. It appears after Slice 3, even though the slice defining condition for Slice 3 implies the slice defining condition for Slice 2. In this case, none of the implementation alternatives associated with Slice 2 will be selected by the adapt operator. If 1,411 Kb/s > bw ≥ 706 Kb/s, then ϕ(Slice 3) will be selected. If the 1,411 Kb/s > bw ≥ 353 Kb/s, then ϕ(Slice 3) will once again be selected by the adapt operator. We can safely eliminate all implementation alternatives associated with a slice, if they constitute dead code for all slice orders in Σ.

We can also analyze the transition graph to check for unreachable nodes and dead-ends. An unreachable node is one that has no inbound arcs in the transition graph. A dead-end is a node that has no outbound arcs in the transition graph. Different applications may have different constraints for the transition graph. One possible constraint would require that there be a path in the transition graph from each node to all other nodes. This constraint guarantees that all implementation alternatives are reachable from all other implementation alternatives (if the proper sequence of conditions were to occur).

6. Related Work

Many projects have used adaptation to guarantee QoS requirements and have developed languages to define QoS specifications. The Quasar project [2, 8, 9] developed a networked video and audio player that uses feedback to dynamically adjust to available bandwidth, latency, latency variation, and synchronization of audio and video. The system allows a user to specify QoS adaptation requirements using a micro-language, which gives users the ability to specify weights and coefficients as in the Adaptive Audio Quality application [2]. Staehli *et al.* [10] describes a methodology for QoS specification and provides a formal definition of presentation quality.

There has been a large amount of discussion on issues concerning adaptive systems. McIlhagga *et al.* [11] discuss common aspects and abstractions of adaptive applications in mobile and distributed environments. They suggest that in designing adaptive software, attention should be focussed on defining an implementation space (a set of implementation alternatives), the resource usage to monitor, and the required availability of resources that cause switches between implementations as well as possible degradation paths (i.e., transition paths). Adaptation spaces provide a formal, generic framework that clearly includes all of these dimensions. Further, adaptation spaces enable the implementation of generic components for adaptation at run time.

Shrobe [5] discusses the importance of adaptation in achieving survivability in large-scale information systems. Berman and Gray [12] define an adaptibility metric, which measures an algorithm's ability to assure correct behavior for all possible cases and to guarantee efficient performance in each case.

Two approaches have developed frameworks and specification languages for adaptive systems: BBN's Quality of Service for CORBA Objects (QuO) [13, 14] and the Real-Time Software Adaptation System (RESAS) [15]. The QuO architecture provides QoS support for CORBA objects and QuO extends CORBA by providing a description language to specify QoS requirements for object connections. RESAS provides a system of software models and mechanisms for writing and dynamically adapting software for performance and reliability. RESAS separates the specification of adaptation from the implementation and uses adaptation algorithm specifications to describe when and how to adapt to the current state of the application.

Although similar, the adaptation space formalism presented here is a more abstract model of adaptation than the QuO description language and the RESAS adaptation algorithm specifications. By using adaptation spaces, it is possible to perform certain kinds of formal reasoning (e.g., guaranteeing correct adaptive decisions) on adaptive applications. Neither QuO nor RESAS provides this ability because the adaptive requirements are not separated from the application specification. The adaptation space model also provides a mechanism to describe the overall adaptivity of systems without regard to domain specifics. Adaptation spaces do not rely on specific implementations of components; rather, adaptation spaces generalize the notion of implementation as an implementation alternative. Adaptation spaces can be used for a wider range of adaptive software applications than the QuO architecture and RESAS. Furthermore, adaptation spaces provide a model that highlights preference information and software designer involvement in preference specification, which is not available in QuO or RESAS. The use of preference in adaptation spaces explicitly supports policy. That is, adaptation spaces provide a generic mechanism and a formalism to implement policy based adaptation algorithms, while ensuring feasibility constraints. The application is free to specify additional information about each implementation alternative and use appropriate algorithm(s) or heuristic(s) to reflect the desired, application-specific policy. The adapt operator needs to know only the current preference order each time it is called. The Adaptive Audio Quality example provides a simple example of an application-specific mechanism for specifying preference using coefficients and weights with an algorithm for computing rank.

The related work demonstrates a growing interest in adaptation. We have a generic framework for adaptation that has been successfully applied in applications that adapt to support QoS as well as applications that adapt for survivability. Our adaptation space architecture is more abstract and hence more generic than the frameworks of related work discussed above. Finally, the adaptation space formalism supports reasoning and analysis of adaptation.

7. Conclusions and Future Work

From the beginning, our work on adaptation spaces has been driven by the goal of supporting application-specific adaptation within a single, generic framework. Furthermore, we consider adaptation as a core mechanism, regardless of the reasons for adaptation, e.g., to improve performance, to provide appropriate responses to guarantee quality of service, to move in and out of various defense postures, and so forth.

This combination of being both application-specific and generic lies at the heart of our contribution. We have focused on the fundamental nature of adaptation and we have consistently distinguished the application-specific parts (such as the mechanism used to set preference) and the generic parts (such as the adapt operator that respects the preference configuration).

Our contributions to date are built on the following key insights:

- The representation of conditions (in the condition space) is distinct from the representation of implementation alternatives (in the transition graph).
- We can represent arbitrary policies (used to select implementation alternatives) through the single abstraction of preference.
- Feasibility, based on conditions, is distinct from and orthogonal to preference.
- The use of slices characterized by minimum conditions provides reasonable flexibility in ex-

pressing feasibility while allowing the feasibility check to be tractable.

Given that we have defined and formalized what it means to be an adaptation space and how the next implementation alternative is selected (using the adapt operator), we plan to extend adaptation spaces to accommodate composition. We have two kinds of composition in mind: multiple (sibling) components adapting in parallel and adaptation at multiple levels of granularity. We want to further apply the adaptation space formalism to the problem of formal analysis of survivable systems. Finally, we want to create a graphical tool to fully support adaptation space specification and analysis.

8. References

[1] "Extensible Markup Language (XML) 1.0," in *W3C Recommendation 10-February-1998*: World Wide Web Consortium, 1998.

[2] J. Walpole, C. Krasic, L. Liu, D. Maier, C. Pu, D. McNamee, and D. C. Steere, "Quality of Service Semantics for Multimedia Database Systems," presented at Database Semantics (DS-8), Rotorua, New Zealand, 1999.

[3] A. Le Meur, "A Model for Building Tailorable Adaptive Multimedia Applications," Oregon Graduate Institute of Science and Engineering, Technical Report.

[4] C. Cowan, C. Pu, and H. Hinton, "Death, Taxes, and Imperfact Software: Surviving the Inevitable," presented at 1998 Workshop on New Security Paradigms (NSPW), Charlottesville, VA, 1998.

[5] H. E. Shrobe, "Two Challenging Domains," *ACM Computing Surveys*, vol. 28, pp. 12-es, 1996.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," presented at The 7th {USENIX} Security Symposium (SECURITY-98), Berkeley, California, 1998.

[7] H. M. Hinton, C. Cowan, L. Delcambre, and S. Bowers, "SAM: Security Adaptation Manager," presented at Annual Computer Security Applications Conference (ACSAC), Phoenix, AZ, 1999.

[8] C. Cowan, S. Cen, J. Walpole, and C. Pu, "Adaptive Methods for Distributed Video Presentation," *ACM Computing Surveys*, vol. 27, pp. 580-583, 1995.

[9] S. Cen, C. Cowan, R. Koster, D. Maier, D. McNamee, C. Pu, D. C. Steere, J. Walpole, and L. Yu, "A Player for Adaptive MPEG Video Streaming Over the Internet," presented at 26th Applied Imagery Pattern Recognition Workshop (AIPR '97, SPIE), Washington, D.C., 1997.

[10] R. Staehli, J. Walpole, and D. Maier, "Quality of Service Specifications for Multimedia Presentations," *Multimedia Systems*, vol. 3, pp. 251-263, 1995.

[11] M. McIlhagga, A. Light, and I. Wakeman, "Towards a Design Methodology for Adaptive Applications," presented at The 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98), Dallas, Texas, 1998.

[12] P. Berman and J. A. Garay, "Adaptability and the Usefulness of Hints (Extended Abstract)," presented at Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, 1998.

[13] J. Zinky, D. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, pp. 1-20, 1997.

[14] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," presented at The IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, 1998.

[15] T. E. Bihari and K. Schwan, "Dynamic Adaption of Real-Time Software," *ACM Transactions on Computer Systems*, vol. 9, pp. 143 - 174, 1991.