Submitted for publication

Dynamic Architecture-Based Monitoring

David Garlan Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
{garlan,schmerl}@cs.cmu.edu

Jichuan Chang
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI
chang@cs.wisc.edu

Abstract

Increasingly, systems need to have the ability to adapt at runtime. In the past, system adaptation has been coded into each application, making it costly to modify the adaptation policy or mechanism, difficult to reuse in other applications, hard to reason about, and obfuscates the primary application code. In this paper we outline an approach that addresses these problems by externalizing adaptation mechanisms. Specifically, we describe mechanisms for observing a running system, interpreting these observations according to the system's architecture, analyzing the system architecture to ascertain if it still meets its design constraints, triggering repair strategies to repair the architecture if not, and applying these architectural changes to the running system. We discuss the design of such a system, and focusing on our research into providing a monitoring infrastructure that can be independently applied to a variety of systems. We also describe our tool support for interpreting and displaying observations according to an architectural description of the system written in Acme.

1 Introduction

An increasingly important requirement for systems is the ability to adapt at run time to handle such things as resource variability, changing user needs, and system faults. Sometimes such adaptation can be user-assisted – for example a user (or system administrator) might load new applications, change various system parameters, or reconfigure the load on a set of servers. But more and more, because of their inherent complexity and the need for rapid adaptation response, systems must handle their *own* adaptation. An important engineering question then becomes how should one add adaptability mechanisms to complex software-based systems.

In the past, system adaptation has largely been handled internally to the application. For example, applications typically use generic mechanisms such as exception handling, or heartbeat mechanisms to trigger application-specific responses to an observed fault. For other aspects of adaptation, such as resource-based adaptation, systems typically wire in application-specific policies and mechanisms. For example, a video teleconferencing application may decide how to reduce its fidelity when transmission bandwidth is low using some combination of compression and reduced framesize and resolution.

While internal adaptation can certainly be made to work, it has a number of serious problems. First, when adaptation is intertwined with application code it is difficult and costly to make changes in adaptation policy and mechanism. Second, for similar reasons, it is hard to reuse adaptation mechanisms from one application to another. Third, it is difficult to reason about the correctness of a given adaptation mechanism, because one must also consider all of the application-specific functionality at the same time.

An alternative approach is to develop *externalized* adaptation mechanisms. For such systems, adaptation is handled outside of the application. Systems are monitored for various attributes, such as resource utilization, reliability, delivered quality of service. Based on that monitored information, external mechanisms decide whether the application must be reconfigured.

In contrast to the internal approach, externalized adaptation has many benefits: adaptation mechanisms can be more easily extended; they can be studied and reasoned about independently of the monitored applications; they can exploit shared monitoring and adaptation infrastructure.

However, there are a number of hard questions that one must then answer. How does one obtain the necessary information about the monitored system? How can one make intelligent decisions about when to make an adaptation, and how it should be made? How can one take into account application semantics that might favor or prohibit certain kinds of adaptations? What kind of infrastructure for adaptation will permit diversity, analyzability, extension, and reuse – all in the face of complex, distributed systems?

In this paper we provide answers to some of these questions. Specifically, we describe an approach in which architectural models are used as the basis for automated adaptation. Low-level implementation-specific behavior is obtained from a system of "probes."[1] This low level information is then interpreted by a system of "gauges" in terms of their architectural significance. Decisions can then be made at the architectural level of abstraction about how to respond to observed anomalies.

2 Dynamic Architecture-Based Adaptation

2.1 Requirements

To take advantage of the insights gained through high level models of the target system, three pieces of system infrastructure are essential.

- **Monitoring**: There must be some way to observe the behavior of a running system. Without such information, it will not be possible to determine whether a change needs to be made. The infrastructure must support a wide variety of monitoring technologies. Given our desire to monitor a wide variety of applications, and that it is likely that these applications will be written in a variety of languages, different monitoring technologies are likely to be used. For example, it should be possible for the infrastructure to take advantage of monitoring technologies such as source code instrumentation to produce dynamic information, replacement of specific libraries with monitoring versions of those libraries, or services that give the state of the environment in which a program is running.
- **Interpretation**: The information that is being gathered must be interpreted in the context of the system properties or features described in the high-level model. For example, a monitored value that measures the bandwidth between two nodes on a network may not have an appropriate meaning until its effect of the system (as expressed in the high-level model) can be determined (for example, in terms of completion or response time).
- **Reconfiguration**: Once a value is interpreted in the context of the model, it will be possible to detect whether the system is running correctly, or within acceptable parameters, and whether a change is needed. If a change is necessary, the target system is changed. In most cases, this will mean adapting the high-level model and ascertaining the correctness of the adaptation at that level before propagating the change to the implementation.

Given these three pieces, the following questions need to be addressed by systems providing dynamic adaptation:

- Monitoring: What is the appropriate technology for providing monitoring of an application? How can support for a wide variety of monitoring techniques be provided? How can monitoring be added to existing systems in a way that minimized the effect on the system?
- Interpretation: What kinds of high-level models are to be supported? How and when should this interpretation commence, and how often should it be made?
- Reconfiguration: What technologies should be used to determine if interpretations result in an incorrect system? How do we choose a particular reconfiguration that best addresses the

[1] In this paper, we use the terminology of the DASADA program. DASADA (Dynamic Assembly for Systems Adaptation, Dependability, and Assurance) is a DARPA funded research program comprising numerous researchers from U.S. universities and research companies.

errors detected in the system? How often, and under what conditions, should reconfiguration be applied? How is the reconfiguration effected in the running system?

2.2 Architectural Models

One of the central issues in answering these questions is determining what kinds of models should be used to interpret observed behavior. In principal many kinds of models might be used, including behavioral, performance, structural, and timing models.

In our work we are focusing on the use of architectural models as the vehicle for monitoring, interpretation, and reconfiguration. An architectural model represents a system in terms of its principal run time parts ("components"), and their pathways of communication ("connectors"). In addition, many architectural models include semantic detail that explains such things as expected properties of the components or connectors, constraints on topology or behavior, and allowable forms of evolution.

Architectural models are particularly useful for many kinds of automated system monitoring and adaptation. First, they provide a suitably high level of abstraction for interpreting low-level behavior in terms that an engineer or user can appreciate. For example, low-level details such as network routing pathways can be encapsulated as a single connector with values for throughput, latency, and congestion. These in turn can be interpreted in terms that are important to a user – such as transmission time between two parts of a system. Second, they allow one to make explicit constraints that can be evaluated against actual behavior. For example, when one associates explicit protocols of interaction for a connector [1], tools can check that actual communication conforms to the expected protocol. Third, architectural models are often close enough to implementation structures that one can map architectural reconfiguration decisions to implementation reconfiguration commands.

2.3 Integration Framework

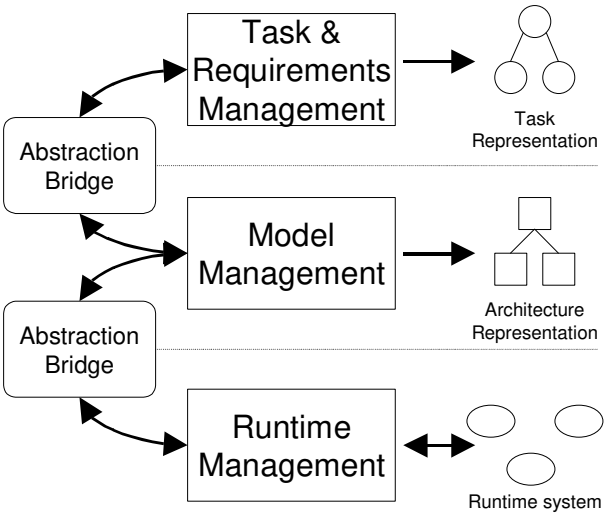

Figure 1. A framework for automated system adaptation

Our approach is based on the layered approach illustrated in Figure 1. This figure is a generic three-layer framework on which our approach is based. Each layer consists of a set of representations, together with a manager responsible for monitoring and adapting the representations at that level. Inter-level interactions are facilitated by components, called

abstraction bridges. This framework provides the context in which architectural models are used in dynamic adaptation.

The lowest level is responsible for monitoring a system's runtime properties. It could consist of the system itself, together with its operating environment (networks, processors, I/O devices, communications links, etc.). The *Runtime Management layer* is responsible for monitoring system behavior and for making changes to the system or its environment. For example, it might observe network or processor load, possibly affecting those quantities by changing processor assignments or network routing schemes. Observer runtime information is propagated upwards through the abstraction bridge, which condenses, filters, and abstracts those observations in order to render that information in architecture-relevant terms.

The middle layer is responsible for automated support for system adaptation. It forms the centerpiece of the approach, consisting of one or more architectural models of the system, together with *Model Managers* that determine when information from the lower level violates architectural design assumptions. In addition, the model management layer is responsible for determining repair strategies, and for communicating the corresponding control/reconfiguration operations to the lower level. For example, overload of some communication link (observed by the runtime manager) might violate minimal expectations of bandwidth of some connector in the architecture. An architecture manager, having detected the problem, might then change the connection mechanism to use another channel or communication protocol. Problems that cannot be handle automatically are propagated up to the next layer.

The top level is responsible for managing the users' higher-level requirements. It maintains a representation of users' tasks, it propagates system requirements down to the architectural modeling layer, and it handles unrecoverable problems from below. This top layer is crucial: without it, the system would not be able to determine the values of specific operating parameters of architectural models, or be able to respond to users' changing needs. For example, deciding whether a slow communication link violates an architectural assumption will often depend on what the user intends to do with the data communicated over it.

As a general framework, the scheme just described could be instantiated in many ways depending on the technical details of each part. At the lowest level, one could support many kinds of system observation: behavior (in the form of event traces), performance monitoring, intrusion detection, and real-time behavior are a few of the possibilities. At the middle layer, one could use any number of architectural models and architectural analysis capabilities. At the top level, on might have many ways of representing user tasks, including agents, process models, and workflow descriptions.

Although this framework is very generic, we are instantiating it with a particular set of technologies. In particular, we are using the following technologies to investigate and facilitate dynamic adaptation:

- **Level 1 – Performance Monitoring, Abstraction, and Reconfiguration**: In order to be able to adapt to their execution environment, applications need information about the status and expected performance properties of the environment. Because of its performance for complex distributed applications, there has been considerable interest in monitoring tools that support information gathering for adaptive application. Not surprisingly, this is a challenging task. How is it possible to collect detailed information without introducing significant overhead? Moreover, applications may not care about load conditions *per se*. Instead, they would like an estimate of future performance (running time or throughput), which means that future performance must be extrapolated from past load conditions. We are using Remos [9], a Resource Monitoring System, that collects information on host and network loads and makes this information available through an explicit API.
- **Level 2 – Architectural Modeling, Detection of Constraint Violations, and Automated Architectural Repair**: We are using the Acme [6] architecture description

language as the basis for specifying architectural designs, and design constraints, and are adapting this language to support dynamic monitoring and adaptation. We are also using AcmeStudio, and design environment for constructing Acme designs, as the basis for our runtime tool support. AcmeStudio has been modified to make architectural models available at runtime. Furthermore, we are planning to adapt AcmeStudio to perform runtime architectural constraint analysis, previously provided statically via the Armani extensions to the Acme language. In the area of architectural repair, we plan to extend Acme with facilities to specify architectural alternatives, and repair strategies that provide a principled mapping from an existing architecture to a new, corrected architecture in the event of a runtime error. These mappings can be analyzed statically to determine that they result in correctly performing architectures.

- **Level 3 – Task Management**: We plan to take advantage of existing strategies for representing tasks (such as workflow and process models, agents) and treat them as low-level task representations into which an even higher level of task description can be compiled. We plan to develop a mapping between this level and the architecture level that translates task changes into architectural changes.

The above generic framework provides an overall view of our approach to dynamic architecture-based adaptation. The remainder of this paper concentrates on providing support for architecture-based monitoring, and in essence focuses on the abstraction bridge between the runtime level and the architectural level.

3 Gauges for Dynamic, Architecture-based Monitoring

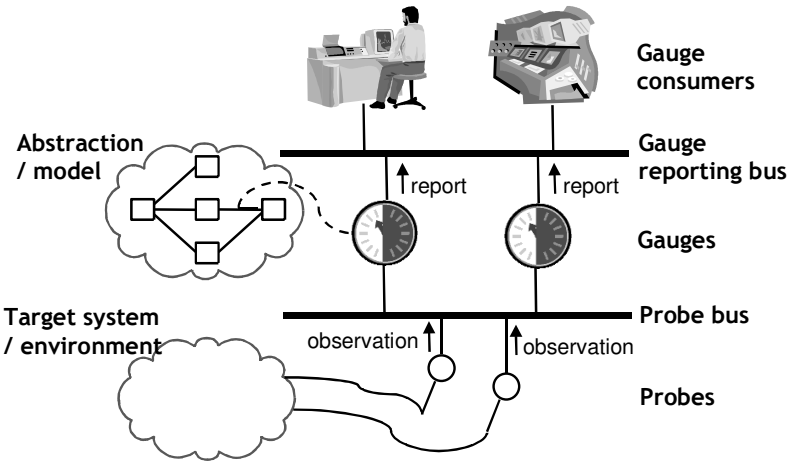

Figure 2. The Gauge Infrastructure Architecture

The Gauge Infrastructure is solely concerned with monitoring the running system. It provides a bridge between the low-level observations of probes and the high-level interpretations of system activity according to an Acme description. The overall conceptual architecture is illustrated in Figure 2. At the lowest level is a set of *probes*, which are "deployed" in the target system or physical environment, and announce observations of the actual system via a "probe reporting bus." At the second level a set of *gauges* consume and interpret lower-level probe measurements in terms of higher-level model properties. Like probes, gauges disseminate information via a "gauge reporting bus." The top-level entities in Figure 2 are *gauge consumers*, which consume information disseminated by gauges. Such information can be used, for example, to update an abstraction/model, to make system repair decisions, to display warnings and alerts to system users, or to show the current status of the running system; thus entities at the Model Management layer of the integration framework will form the set of possible gauge consumers.

Gauges are decoupled from the implemented system by inserting a layer that deals with probes. This decoupling gives the opportunity to run gauges in a distributed fashion, so that they do not affect the performance of the system being gauged, and also gives the opportunity to use probe reports in a variety of ways for different models (for example, the same probes outputs could be used by gauges attached to other models or types of models, such as Meta-H or UML).

3.1 An Illustrative Example

To illustrate how this conceptual architecture can be realized in practice consider Figure 3, which presents a simple example of probing and gauging. Imagine that we have a target system that consists of a *sender* that is sending files to a *server*. The architectural model of this system, represented at the top of the figure, consists of two components (the *sender* and the *server*) and one connector, *L*, representing the network link between them. The implementation of this system consists of the programs comprising the sender and server (these could be further elaborated, but that is of no interest in this example), the actual network links between the machines on which the sender and server are executing, and the set of files to be delivered. The user of this system requires that the set of files should reach the server within a certain deadline. Whether this deadline is being met by the running application depends on the size of the files to be transferred and the bandwidth available between the sender and the receiver. Thus, to ascertain the behavior of the system with respect to this performance attribute, we need to insert some probes and gauges.

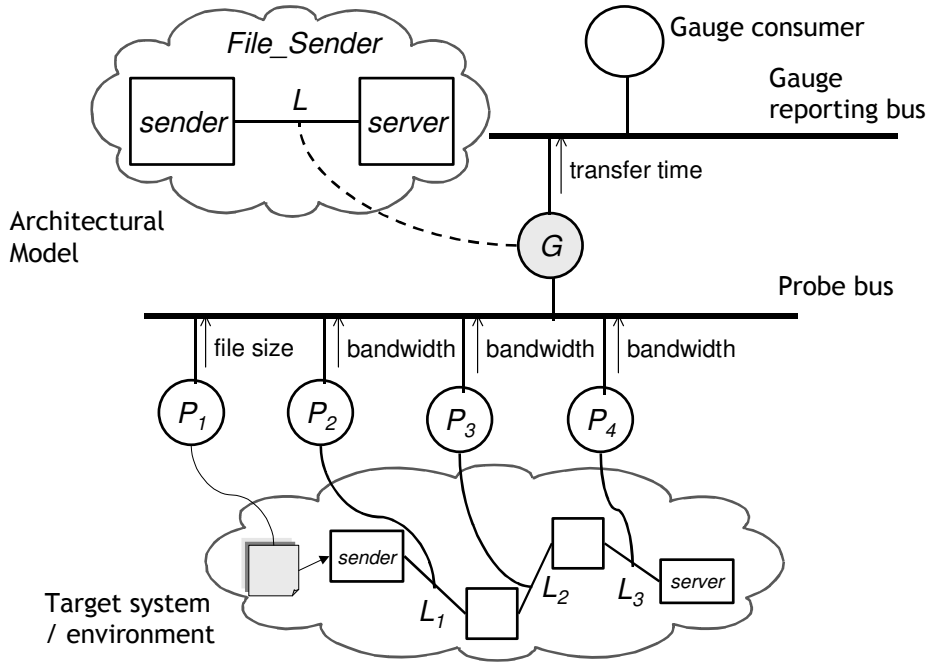

Figure 3. Gauge example

Two types of probes are inserted in the target system. One type of probe monitors the environment, and reports the bandwidth of the various links between the machines of interest; these probes are represented by P_2, P_3, and P_4. The second type of probe, P_1, is inserted into the sender, and reports the size of the files being loaded into the system. Such a probe might be realized through instrumenting the system call *fopen*, for example. This probe information is not directly related to the performance attribute in the architectural model, which is in terms of transfer time between the *sender* and the *server*. To achieve this level of monitoring, a gauge is

attached to the connector L in the architectural model (the details of attachment are discussed Section 4.1). This gauge uses the probe values and calculates the estimated transfer time based on the file size and the available bandwidth. This value is then reported as the transfer time of that particular connector, to be consumed by a monitoring tool that will evaluate whether the deadline can be met.

The nature of probes, technologies for inserting them into systems, and how they report values is not discussed in detail in this paper. However, their context with respect to gauges is important in highlighting the difference between the low level, system observations and the high level, architectural observations that gauges produce.

3.2 Gauge Definition

Gauges are software entities that gather, aggregate, compute, analyze, disseminate and/or visualize measurement information about software systems. Software tools/agents, software engineers and system operators consume such information, use it to evaluate system state and dynamically make adaptation decisions. In its pure form, a gauge does not change its associated model or control the software system directly. However, the outputs of a gauge may be used by other entities to effect such changes.

Several principles or assumptions underlie our notion of gauges and have been used to guide the design of the gauge specification and gauge APIs. These assumptions include:

1. *The value (or values) reported by a gauge can have multiple consumers*. A single gauge consumer can use multiple gauges. For example, there may be gauge consumers that simply monitor and report values to the user, and other consumers that automatically detect impending failure and take action to adapt the underlying system automatically, but use the same model as the basis for both activities. In this case, we do not want to duplicate gauges.
2. *Different parties will develop different types of gauges.* We expect there to be a wide variety of gauge types, reflecting the diverse needs for system monitoring and adaptation. We expect that in many cases a heterogeneous mix of gauges will be operating in a distributed fashion on multiple (heterogeneous) platforms.
3. *The set of gauge consumers can change dynamically.* In this way we can dynamically adapt our monitoring infrastructure to add new observational capabilities as needed.
4. *Each gauge has a type, which describes the gauge's setup and configuration requirements, and the types of values that it reports.* Gauge developers and gauge consumers should have a contract that specifies what to provide and require from a gauge.
5. *Gauges are associated with models.* Models allow gauges to interpret their inputs and produce higher-level outputs. Moreover, gauge values must be meaningful in some context, and the model provides the context. For example, the transfer time gauge of the example above interprets the physical observations in terms of an abstract connector in the context of a specific architectural model.

We also identify the need for certain gauge administrative entities – called *gauge managers* – that will be developed to facilitate the control, management, and meta-information query of gauges.

Given the diversity of gauges, implemented by many different parties, using different programming languages, running on different hardware and software platforms, it is important to be able to characterize gauges so that a system builder can determine what types of gauges are available and what kinds of capabilities that type of gauge has. Gauge developers could also use such a characterization as a functional specification around which to base their implementations, and by the gauge run-time infrastructure to manage gauges by providing gauge meta-information.

In this section we consider how one might specify a gauge. In brief, a gauge's specification describes (1) its associated model (and model type), (2) the types of values that it reports and the associated model properties, and (3) setup and configuration parameters.

Each gauge has a type. A gauge *type specification* describes the shared features of instances of a gauge type. A gauge *instance specification* defines a particular gauge. A gauge instance includes information about the gauge that elaborates the gauge type specification and associates the outputs of the gauge with a particular abstract model or elements of a model. For example an instance of the transfer time gauge type, illustrated in Figure 3, would identify the IP address set-up parameters, a default "frequency of sampling" control parameter, and indicate the model and connector for which it is calculating the transfer time value.

A gauge type specification is a tuple consisting of the following parts:

1. The name of the gauge type: for example, *XferTime_Gauge_T*;
2. The set of values reported by the gauge (specified using a name and a type): for example, the *XferTime_Gauge_T* reports one value, *xferTime* of type *float*;
3. Setup parameters (including name, type, and default value for each parameter): for example, the *XferTime_Gauge_T* has two setup parameters: *Src_IP_Addr* and *Dst_IP_Addr*, which are both of type *String* and have no default value;
4. Configuration parameters (including name, type, and default value for each parameter): for example, the *XferTime_Gauge_T* has one configuration parameter *Sampling_Frequency*, which is of type *milliseconds* with a default value of *50.* The sets of configuration parameters and setup parameters are not necessarily disjoint. A default value should be provided for each configuration parameter that is not in the set of setup parameters.[2]
5. Comments: these explain in more detail what a gauge does and how to interpret the values (the values' units, accuracies, etc.) that provide more detail about the functionality of the gauge.

Table 1 describes a gauge type for the gauge G in Figure 3, that measures the transfer time value in milliseconds, represented as a floating point number.

Gauge Type	`XferTime_Gauge_T`
Reported Values	`xferTime: float`
Setup Parameters	`Src_IP_Addr: String [default=""]` `Dst_IP_Addr: String [default=""]`
Configuration Parameters	`Sampling_Frequency: int [default=50]`
Comments	`Latency_Gauge_T measures network latency of a connector whose endpoints are defined by a source and destination IP address.`

Table 1. An example of gauge type specification

A gauge instance specification is a tuple consisting of the following parts:

1. The name and type of the gauge instance: for example, *G* is the name of the latency gauge in Figure 3, which is of gauge type *XferTime_Gauge_T*;
2. The name and type of the model that the gauge is associate with: for example, *G* is associated with a model called *File_Sender*, which is of type *Acme*;
3. Mappings from values reported by the gauge to the associated model properties. Each mapping is a tuple of *<GaugeValue, ModelProperty>*, meaning that the *GaugeValue* actually reflects the value of *ModelProperty*: for example, the mapping for *G* is *<xferTime, L.xferTime>*;

[2] Currently only literal values are allowed for setup and configuration parameters.

4. Setup values: these can be statically specified or dynamically provided upon gauge creation. If no value is provided, the default value of this gauge type should be used;
5. Configuration values: these can be statically specified or provided at run-time. If no value is provided when the gauge is created, the default value for this gauge type should be used.
6. Comments: to describe more details of the gauge's function.

Table 2 specifies the gauge instance *G* that we discussed in the previous example.

Gauge Name: Gauge Type	`G: XferTime_Gauge_T`
Model Name: Model Type	`File_Sender : Acme`
Mapping	`<xferTime, L.xferTime>`
Setup Values	`Src_IP_Addr = L.src.IP1;` `Dst_IP_Addr = L.snk.IP2;`
Configuration Values	`Sample_Frequency = 100`
Comments	`G is associated with the L Connector of the system, File_Sender, defined as an Acme model.`

Table 2. An example of gauge instance specification

The above definition of gauges is very general, can be applied to a wide variety of monitoring needs, and is not restricted to being associated with one particular model or modeling language. This is an advantage over other approaches, which restrict monitoring to one particular system or one particular architectural style.

4 Implementation

To this point we have described generally the way that gauges are specified, and how they are used to monitor a system. Given this general infrastructure, we have experimented with a set of tools and techniques that allow monitoring in the context of Acme models. This section describes the state of our implementation.

4.1 Attaching Gauges to Acme

As indicated earlier, gauges are used to interpret observations of the running system in the context of an architectural model. These observations form part of the semantics of the system and therefore should be mapped to the semantics of the architecture. Acme is a general-purpose architecture description language that is style-independent. Although particular styles can be defined in the Acme language, the building blocks of an architecture are generic components and connectors, with associated properties that do not have any inherent meaning. Styles are defined by specifying particular properties to be associated with particular types of components, and also in defining constraints that can be used to do some semantic analysis of the style. Furthermore, analysis tools can analyze certain properties in an architecture to arrive at some conclusion about the correctness of the architecture according to the analysis.

Because the semantics of an architecture are captured in the property mechanism of Acme, gauges are attached to Acme properties. The meaning of this is that the value(s) reported by a gauge are actually values of the properties to which they are attached. In this way, analysis tools can use these changing properties. For example, we anticipate that design constraints over the properties in an architecture will be re-evaluated when a property value changes dynamically as reported by a gauge. This will also allow any other tools that analyze Acme properties to be used dynamically. Attaching gauges to properties also means that tools that currently work with Acme descriptions will not need to change when gauges are added.

```
connector L = {
    role src {
        property ip : string;
    };
    role snk {
        property ip : string;
    };
    property xferTime : float
        <<  gauged : boolean = true;
            gauge : XferTime_Gauge_Spec = [
                name = "G";
                type = XferTime_Gauge_T;
                value = xferTime;
                src_ip_addr = src.ip;
                dest_ip_addr = snk.ip;
            ];
        >>;
};
```

Figure 4. A Gauge Attached to an Acme connector.

Attaching a gauge value to an Acme property is done by associating a *meta-property*, describing the gauge, with the property. Meta-properties are currently used in Acme to assign details like default values or units of measure. Figure 4 shows an Acme description of the connector *L* from Figure 3. The fact that a property is a gauged value (and therefore its value is assigned at runtime) is set by having the meta-property *gauged* associated with the property. The next meta-property (*gauge*) defines the name and type of the gauge, which gauge value is mapped to this particular property, and the setup and configuration parameters. The Acme gauge specification in Figure 4 corresponds to the gauge instance specification in Table 2. The *gauge* meta-property is an Acme record that is defined elsewhere in the Acme description. Each gauge type has a corresponding Acme record type. These records can be generated automatically from the gauge type specification.

4.2 Tool Support for Gauges

Based on the definition of gauges given above, we have developed an implementation of the gauge infrastructure that provides a set of Java classes and interfaces, and uses the Siena wide-area event notification system [3] as the communication substrate through which events are communicated between gauges and their consumers. The class hierarchy for this implementation is presented in Figure 5. The classes provided by the infrastructure are shown in the middle of the figure; the interfaces that need to be implemented for particular gauges or gauge consumers are at the top of the figure. This implementation hides the communication mechanism used to send the events. In fact, we have one implementation that uses Siena, and another that transparently utilizes Java RMI to transport events – in either case, the code that the Gauge Developer or Gauge Consumer Developer has to write is exactly the same, allowing portability across communication mechanism.

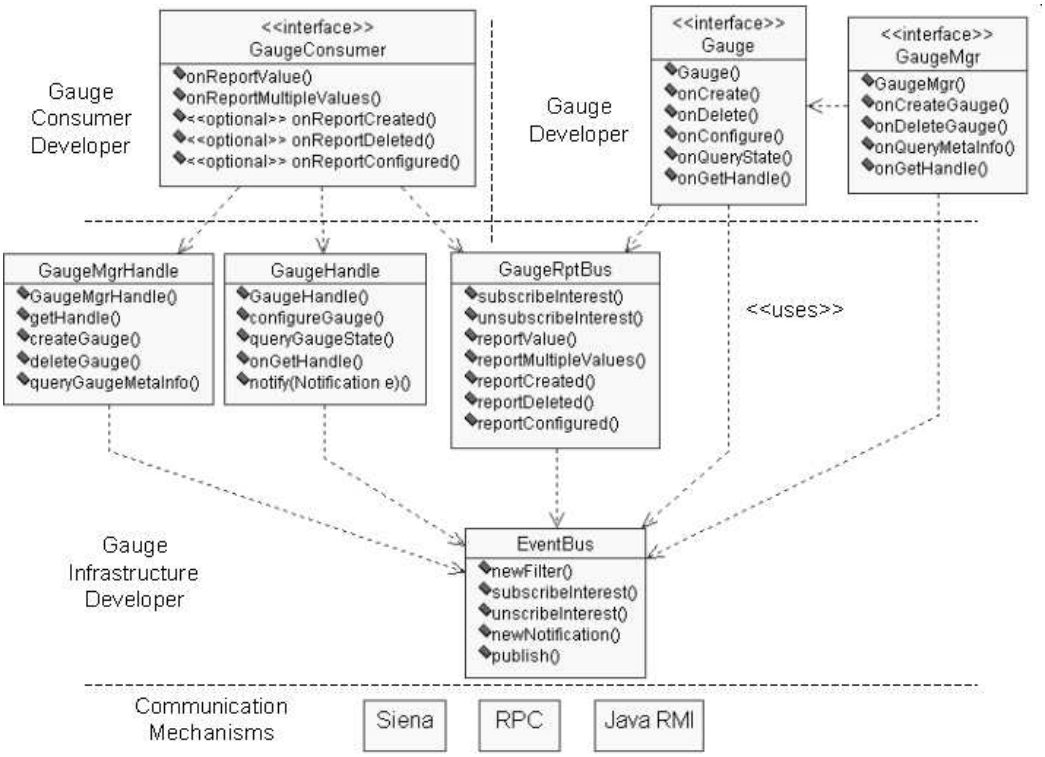

Figure 5. Gauge Infrastructure implementation

An Acme description with a set of attached gauges can be used to generate the gauge instances so that consumers can listen to those messages. We have written a tool that does this, and also generates the necessary code to connect with our design environment, AcmeStudio, which can be used to display the gauge outputs dynamically.[3] Figure 6 shows the process by which this is achieved. The *Gauge Generator* takes the Acme file and produces a *Monitoring Tool*. This tool, when executed, will create and configure the gauge instances (by connecting with the appropriate gauge managers), and uses the COM interface of AcmeStudio to load the Acme description, start listening for gauge values, and propagate these values to AcmeStudio.

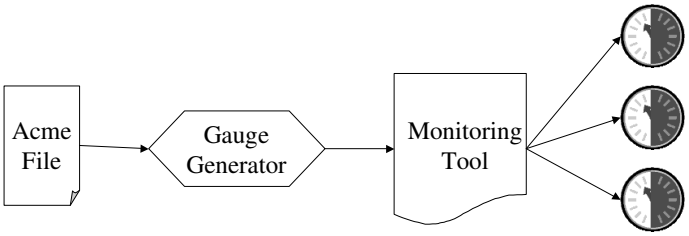

Figure 6. Generating gauges from Acme descriptions

Figure 7 illustrates an instance of the gauge infrastructure with associated tools after the tool described above is run. The tool first creates the gauge, in the center of the figure, and sets it up so that it is reporting its value in the context of the system's architecture (it is attached to the *xferTime* property of the connector *L*). Once this gauge is created, it activates various probes in

[3] This tool is currently being implemented and will be ready by the time camera-ready copies of the paper are due.

the target system. Remos, a network service that gives applications access to the available bandwidth, is used to report the bandwidth between the sender and the server. The Remos probe is focused so that it reports information for the network between the hosts described in the *ip* property of the roles in the connector *L*. Currently, the filesize probe is hardwired into the sender; more disciplined approaches to deploying and activating probes are being developed by others, and will be used in the implementation of the gauges as they become available. The tool also connects to AcmeStudio, via a COM interface, and loads the description to which the gauge is attached. As it receives reports from the gauge, it changes the associated property via this interface, and AcmeStudio displays the value changing as it is reported.

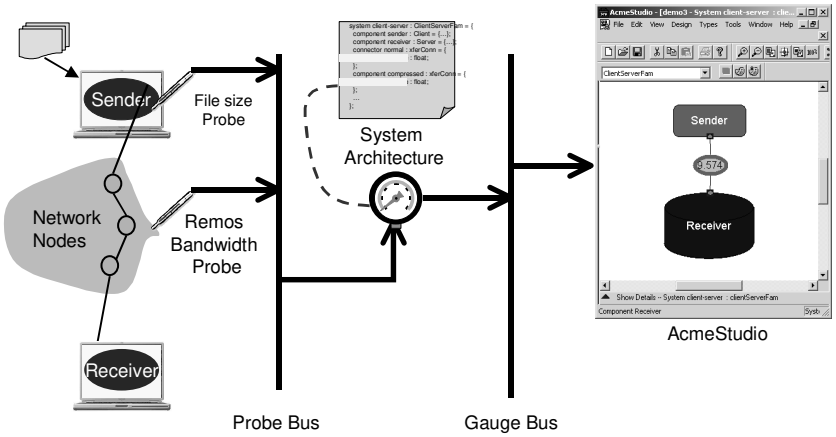

Figure 7. Implementation of monitoring described in Figure 3.

5 Related Work

There is research that has been done into dynamically adapting systems, both for internal and external adaptation. In the internal realm, there are a multitude of programming languages and libraries that provide dynamic linking and binding mechanisms [7][11][4][8] Systems of this kind allow dynamic change to be programmed on a per-system basis, and do not provide external, reusable mechanisms that can be added to systems in a disciplined manner *per se*, as with our approach.

Some systems have been developed that externalize dynamic change and also provide an architectural level of detail for arguing about an implementation. Systems such as C2 [14][15] and Darwin [13] (including its predecessor, Conic [12]) provide a way for an external agent to change the running program (in the case of C2, ArchShell; in the case of Darwin and Conic, a Configuration Manager) by providing a shell through which reconfiguration commands can be issued. However, these systems rely on implementations that closely resemble the architectural model. In C2, the implementation must adhere to the C2 architectural style. In Darwin and Conic, implementations must adhere to a configuration independence principal where all interactions between modules must be through exit and entry points; modules cannot reference other modules except through these ports. In contrast, our approach does not rely on any particular style and does not require that the implementation be in close correspondence with the model. This will allow us to investigate the use of new architectural styles and take advantage of a number of existing architectural styles in dynamic adaptation.

Furthermore, because we are using a language in which architectural design constraints are easily expressed [10], we will be able to check the conformance of observations of implementations to architectural design assumptions. In addition, we should be able to formally analyze adaptation strategies based on earlier work with Wright [1]. Also, work done in DCDL [17][18] for applying consistency constraints specified in a first-order predicate logic language to

determine consistent configurations of dynamically adapting systems is applicable to our use of Armani in analyzing correct software architecture. DCDL uses these consistency rules to determine the set of existing versions of components will result in consistent configurations.

6 Conclusion and Future Work

In this paper, we have introduced a framework and an implementation that provides a general mechanism for adding monitoring and interpretation capabilities that takes advantage of probes to observe executing programs, gauges to interpret those programs, and a strategy that allows these gauges to be attached to properties in an Acme description. This framework and implementation has the advantage that it is:

- Distributed, so that monitoring and assessment of a running application can occur without needing to consume extensive resources on the machine on which the application being monitored is executing;
- General, in that the monitoring capability is not restricted to a particular architectural or implementation style.

Specifically, we have described our approach to providing externalized dynamic adaptation, and our initial implementations of the monitoring component of this approach. The monitoring component is general enough to allow may "probe" technologies, that observe the state of the implemented system, to be used. The values of these low-level observations can be interpreted at the architectural level by attaching gauges to Acme properties. This approach satisfies the requirements that such a technology by general and distributed. We view this work as a first step for achieving the overall framework described earlier. In our ongoing research we plan to extend our results and evaluate the current design. Our plans for evaluation include:

- Applying different kinds of probing technologies in a principled manner. The Remos probe that we have used to observe low-level information is a probe that monitors the network environment of a distributed application. It does not need to be inserted into an application, and so is non-intrusive. Currently, the other probes described in this paper, is handcoded into the application. We plan to coordinate with other researchers developing different types of probe technologies, to evaluate the generality of our gauges with respect to probes.
- Evaluating the performance overhead and synchronization issues with our performance monitoring technology. Our technology, while measuring the available bandwidth, also affects the bandwidth because it may communicate the report along the very channels it is measuring. We have attempted to minimize this by executing the Remos collector and probe on a separate machine to the ones executing the application. A further problem is the time it takes to report and react to values being communicated. For example, if reporting is slow to reach the architectural level, then, by the time a change is required, the conditions in the environment may have change sufficiently to warrant a different change or require no change at all. The effects of this problem, and ways to alleviate it, are areas of further study.

While we have described in some detail the monitoring component, the harder tasks, that of deciding on a change and actually conducting the change, are yet to be done. We plan to take advantage our previous work in architectural analysis to guide us in how to analyze reports; for this, we will initially use Armani constraints to provide feedback on whether a monitored architecture requires a change. Our next step is to develop design a formalism for taking detected errors and associating them with particular changes that will correct the errors – a component we call repair strategies. We also plan to use and extend the work on Aesop [5] to guide us in taking changes at the architectural level and reflecting them in the implementation.

Acknowledgements

DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616, supports this work. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA. We would like to thank the members of the ABLE group.

References

[1] "A Formal Basis for Architectural Connection," R. J. Allen and D. Garlan, in ACM Transactions on Software Engineering, July 1997.

[2] "Specifying and Analyzing Dynamic Software Architectures," R. J. Allen, R. Douence, and D. Garlan, in *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering* (FASE '98), March 1998.

[3] "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service," A. Carzaniga, D.S. Rosenblum, and A.L. Wolf in *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing* (PODC2000), Portland OR. July, 2000.

[4] "Dynamic Binding in Strongly Typed Programming Languages," R. E. Gantenbein, *Journal of Systems and Software* **14**(1):31—38, 1991.

[5] "Exploiting Style in Architectural Design Environments," D. Garlan, R. Allen, and J. Ockerbloom, *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng*, December 1994.

[6] "Acme: Architectural Description of Component-Based Systems," D. Garlan, R. T. Monroe, and D. Wile, in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman (eds), Cambridge University Press, 2000, pp. 47-68.

[7] "The Java Language Environment: A White Paper," J. Gosling and H. McGilton, Sun Microsystems Computer Company, Mountain View, California, May 1996. Available at http://java.sun.com/docs/white/langenv/.

[8] "An Approach to Genuine Dynamic Linking," W. W. Ho and R. A. Olsson, *Software – Practice and Experience* **21**(4):375—390, 1991.

[9] "A Resource Query Interface for Network-Aware Applications," B. Lowekamp, N. Miller, D. Sutherland, D. Gross, P. Steenkiste, and J. Subhlok, in *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing* (Chicago, IL), July 1998.

[10] "Capturing Software Architecture Design Expertise With Armani," R. Monroe, CMU School of Computer Science Technical Report CMU-CS-98-163. Version 2.3, revised January 2001.

[11] "Exploiting Persistent Linkage in Software Engineering Environments," R. Morrison, R. C. H. Connor, Q. I. Cutts, V. S. Dunstan, and G. N. C. Kirby, *The Computer Journal* **38**(1):1—16, 1995.

[12] "Constructing Distributed Systems in Conic," J. Magee, J. Kramer, and M. Sloman, *IEEE Transactions of Software Engineering* **15**(6), 1989.

[13] "Specifying Distributed Software Architectures," J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, in *Proceedings of 5th European Software Engineering Conference* (ESEC '95), Sitges, September 1995. Also published as Lecture Notes in Computer Science 989, (Springer-Verlag), 1995, pp. 137-153.

[14] "An Architecture-Based Runtime Software Evolution," N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, in the *Proceedings of the International Conference on Software Engineering 1998* (ICSE'98). Kyoto, Japan, April 1998, pp. 11—15.

[15] "On the Role of Software Architectures in Runtime System Reconfiguration," P. Oreizy and R. N. Taylor, in *Proceedings of the International Conference on Configurable Distributed Systems* (ICCDS 4). Annapolis, Maryland, May 4-6, 1998.

[16] "An Architecture-Based Approach to Self-Adaptive Software," P. Oreizy, M. M. Gorlick, R. N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 54—62.

[17] "Consistency Issues in Partially Bound Dynamically Composed Systems," B. R. Schmerl and C. D. Marlin, in *Proceedings of the 1996 Australian Software Engineering Conference* (Melbourne, Australia), pp 183-192. IEEE Computer Society Press, Los Alimitos, California. July 1996.

[18] “Designing Configuration Management Tools for Dynamically Composed Systems,” B. R. Schmerl. Ph.D. Thesis, Department of Computer Science, Flinders University of South Australia, December 1997.