AFRL-IF-RS-TR-2004-276
Final Technical Report
October 2004

EN-GAUGING ARCHITECTURES

Teknowledge Corp.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K517

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-276 has been reviewed and is approved for publication.

APPROVED: /s/
RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 2004	**FINAL** **Jul 00 – Dec 03**

4. TITLE AND SUBTITLE

EN-GAUGING ARCHITECTURES

5. FUNDING NUMBERS

C - F30602-00-C-0200
PE - 62301E
PR - DASA
TA - 00
WU - 16

6. AUTHOR(S)

Robert M. Balzer
David S. Wile

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Teknowledge Corp.
1800 Embarcadero Road
Palo Alto CA 94303-3308

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFTB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-276

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577 Raymond.Liuzzi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The goals of the En-gauging Architectures project were (1) to provide community infrastructure that allows programmers to dynamically place probes and gauges into running systems, and (2) to provide dynamic architecture modeling support, specifically for architecture gauges and reconfiguration. In support of (1), our approach was to abstract the experience gained from the Instrumented Connector technology, which allows complex COTS systems running on the Windows platform to be probed, to identify a common run-time infrastructure for a variety of such probe technologies and the facilities needed for those probes to provide inputs to a broad set of gauges. Several DASADA contractors tested the viability of that design by implementing it for their own probe technologies. In support of (2), we developed a COTS infrastructure for analyzing and manipulating architecture models expressed in the Acme architecture description language. We used PowerPoint as an Acme Design Editor that monitors the actual run-time architecture of a system, reifies it into an Acme architecture model, and animates its dynamic behavior through architecture gauges reflected on the screen as a PowerPoint presentation.

14. SUBJECT TERMS Software Gauges, Software Architecture, Architecture Description Language, Architecture Definition Language, Architecture Style, Architecture Dynamism, Formal Specification Language, Architecture Semantics, Siena, Self-Healing System

15. NUMBER OF PAGES 26

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

Technical Summary

The goal of the En-gauging Architectures project was to create the infrastructure to design and deploy gauges on real distributed systems running on commercial platforms, to monitor their architecture and measure their performance. This dynamic system information is made available to a wide variety of subscribers both automated and human, and used to validate performance, resource requirements, and other selected service qualities and to augment the systems' robustness and responsiveness.

Early computing applications were so starved for memory and precious processor time that every detail used in their construction was "compiled away" if it did not directly affect functionality; in fact, such systems performed well in only very tightly-constrained contexts. Modern systems, lacking the extreme resource constraints of old, need not be as highly tuned to the precise usage context, thereby retaining the potential for robustness and adaptability. Modern systems benefit from two adaptive technologies: (1) the ability to compose systems from reusable modules developed and compiled separately and (2) the ability to distribute computing processes onto autonomous computing nodes. Although these technologies enable the *potential* to adapt performance to widely varying contexts, much of the information important for such performance adaptation is still "compiled out" of modern systems.

Fortunately, determining *when* and *how* to adapt a running system to varying configurations and performance demands – the "Quality of Service (QoS) demands" – can be separated from system functionality. To obtain such information it is necessary to model a system's nominal behavior and compare it to its actual behavior for the system's current configuration. While these models are by nature incomplete, they are adequate for validating and tuning performance. Whenever the system deviates from the model, either the system must be reconfigured to achieve its QoS demands or the resources reapportioned to balance those demands. Modeling the system's nominal behavior enables these validations and adaptations to be separated from the system's functionality and to be supported by an external infrastructure.

The En-gauging Architectures project built such a validation and adaptation infrastructure by developing and deploying the gauges that track the system's dynamic architecture and react to its performance characteristics. Our experience with the Acme architecture description language and its Instrumented Connector technology (both developed under DARPA's Evolutionary Design of Complex Software (EDCS) program) were the foundation of the design to monitor the actual run-time architecture of a system, to reify it into an architecture model, and to publish event notifications to "subscribers" interested in such changes to the architecture. Such subscribers comprise analyzers to determine whether dynamic system constraints are satisfied, simulators to establish the system's nominal behavior benchmark, trackers to respond to differences between nominal and actual, and even GUI animators, potentially evoking a human response to redirect system resources.

We also built on our expertise in integrating DARPA's Quorem QoS Condition Service (QCS), and our Instrumented Connector technology, to provide the infrastructure that enables application designers to design and deploy the gauges needed to measure and validate the running system's performance. Using our Composability Framework Services technology, application engineers were able to decide how and when to use this performance and configuration information for adaptation to affect some QoS demands. In point of fact, these QoS demands were reliability and safety demands in the examples to which the technology was applied.

Although the system we designed was adequate for hand crafting probes and architecture-based gauges for simulating and displaying the state of a running architecture, the technology never matured enough to be deployable by anyone other than ourselves – too much expertise in the intricacies of our PowerPoint Design Editor and the MatLab environment (used for simulation description) was required. Our plan for a follow-on Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) Phase II project was to ameliorate this problem considerably, but the follow-on project never took place. Research towards the proposal for that follow-on led to the notion of an *externalized infrastructure* for the monitoring and maintenance of self-healing properties of running systems, systems not designed specifically for being monitored and performance-enhanced. Hence, in the following description in addition to describing the technology we developed for the support of En-gauging Architectures, we describe our role in the DASADA community consensus proposal for this externalized infrastructure.

Approach

To summarize: The goals of the En-gauging Architectures project were:

(1) To provide community infrastructure that allows programmers to dynamically place probes and gauges into running systems;
(2) To provide dynamic architecture modeling support, specifically for architecture gauges and reconfiguration.

In support of (1) our approach was to abstract the experience gained from the Instrumented Connector technology, which allows complex COTS systems running on the Windows platform to be probed [Balzer and Goldman], to identify a common run-time infrastructure for a variety of such probe technologies and the facilities needed for those probes to provide inputs to a broad set of gauges. Several DASADA contractors tested the viability of that design by implementing it for their own probe technologies.

In support of (2) we developed a COTS infrastructure for analyzing and manipulating architecture models expressed in the Acme architecture description language. We used PowerPoint as an Acme Design Editor that monitors the actual run-time architecture of a system, reifies it into an Acme architecture model, and animates its dynamic behavior through architecture gauges reflected on the screen as a PowerPoint presentation. Architectural change event notifications are published to "subscribers" interested in such changes to the architecture. Such subscribers can analyze the architecture to determine whether dynamic system constraints are satisfied, simulate it to establish the system's nominal behavior, track its actual behavior to respond to differences between nominal and actual, or display architectural properties (through an architecture gauge) to evoke a human or automated response to redirect system resources.

Below we describe the technology we developed in support of each of these activities as well as a language design for describing architecture dynamism and the design considerations needed to form a DASADA community consensus on how architecturally significant events should be handled.

Technical Result: Probe Run-Time Infrastructure

Background

The main design goals for the Probe Run-Time Infrastructure [Balzer ‘01a] were to provide Gauges with a common interface for collecting the probe data they need and (2) to remain independent of the probe technology employed and the location of the system being probed. The

first is handled through common APIs and infrastructure described below, while the second is handled by resting the Probe Run-Time Infrastructure on the Siena Wide-Area Event Notification Service.

The Probe Run-Time Infrastructure is designed to support a wide variety of probe technologies through a common set of interfaces and infrastructure. The reason for this diversity is that no single probe technology can instrument all run-time behavior of all systems (Legacy, COTS, or custom developed) on all platforms. To gain that coverage different probe technologies will have to be employed for different systems and platforms, and in some cases multiple probe technologies will have to be employed on a single system.

In addition to these language and platform dependencies, probe technologies differ on when and how they are incorporated (or bound) into a running system or its environment. Some of these probe technologies are bound to a system during the building of that system (e.g. during compilation or during the composition of a load module), some are bound during the loading of a system, some are bound during execution, and some are bound to its environment during its execution.

To handle these differences, the Probe Run-Time Infrastructure was designed to allow probes to be dynamically deployed (probe code situated at a host for subsequent attachment to specific systems at that site), installed (attached to a specific target system), and activated (probes attached to a specific target system or environment turned on). To the extent that a particular probe technology doesn't support such dynamism, the "adaptor" code that couples it to the Run-Time Infrastructure must either ignore the command (because it has already been accomplished statically) or encode it into an action it can dynamically take (such as activating and deactivating statically placed probes by dynamically setting and unsetting a variable used as a guard on those probes).

Design

The Probe Infrastructure rests on, and is described in terms of, the Siena Wide-Area Event Notification Service so that inter-machine transport of commands and data (in the form of Siena events) is transparently handled by Siena.

These wide-area events may be translated into, and out of, a local intra-host standard (such as CORBA or DCOM) by a platform-specific Gateway to improve Intra-host performance. This optional Gateway would translate from the inter-host event API described here into the intra-host, platform-specific API and vise-versa.

A probe is an individual sensor attached to, or associated with (as a monitor of), a running program. This attachment or association can occur either statically or dynamically. It can sense some portion of the program's, or its environment's, execution and make that data available by issuing events. Gauges are the primary consumer of this low-level probe data. They collect and aggregate this probe data into measurements (i.e. values along some dimension).

Where and how it is attached to a running program or its environment is part of the probe specification. For some probe technologies this placement information may be implicit because it is predefined or implied by the probe definition. For others it may be explicitly kept separate from the probe definition and expressed in terms of either the program's architecture or its implementation. If the probe's placement is expressed in terms of the program's architecture and specified in ACME, an Architecture Name Server will be available to provide a mapping from these ACME architecture specifications to the running implementation of the system.

The events[1] issued by probes are either point events with an event name and an arbitrary set of parameters or a duration event with an event name, an event-start or event-end indicator, and an arbitrary set of parameters (i.e. duration events are modeled as a pair of point events indicating the start and end of the duration event).

Individual probes are organized into coordinated groups called configurations that are operated on as a unit. These probe configurations can be deployed to a site, installed on a running instance of a system at that site, activated and deactivated during execution, uninstalled from a running system, and undeployed from a site.

Probe configurations can also be queried to list the set of events they are capable of generating, and be given a directive set of parameters that influence what data they collect and report. Both the directive set of parameters and the influence those parameters have on the probes are specific to a particular probe configuration.

APIs

All APIs are described as events so that probes can be remotely controlled and the data they produce remotely consumed. The syntax shown below is for publishing the event (using Siena's event publication facility). Those clients who register interest in that event receive each published event.

Deploy(Probe-Configuration-Name, Host, Probe-Configuration-Module)

The Probe-Configuration-Module defining the named Probe-Configuration-Name becomes a probe configuration on the named Host. The module contains all the code and declarations needed to construct instances of the probe configuration. The Adaptor is responsible for performing that construction when the probe configuration is Installed (see below) on a running system or its environment. The effect of a Deploy is persistent on a Host (i.e. no need to redeploy if host is rebooted). Only an UnDeploy (see below) removes this persistent Probe-Configuration-Module definition from Host. In keeping with the design conventions of Siena, the Deploy event will contain a URL for the Probe-Configuration-Module and point-to-point protocols will be used by the recipient to obtain the possibly bulky probe configuration module.

Install(Probe-Configuration-Name, Host, System-Spec)

The already deployed Probe-Configuration-Name on Host is incorporated into the specified system or its environment. The probes defined in the probe configuration are initialized to their deactivated state (i.e. not sensing any execution behavior). The System-Spec can specify an already running system, a named system to be run, the next execution of a named system, or all future executions of a named system. If an immediate or future execution of the named system is specified, then Probe-Configuration-Name is incorporated into that System when it is started on Host (this enables probes to sense startup behavior and corresponds to how statically placed probes would be deployed and installed) and a Installed event identifying that execution instance is signaled. These system execution instances (SystemExID) are persistently unique on a particular machine and can be used to identify that execution instance and all Sensed events generated by its probes. To facilitate interoperation among different probe technologies on a machine, these system execution instances should be generated by the platform specific Gateway rather than by an individual Adaptor.

---

[1]The event structure employed was very simple and in our uses, customized to the application at hand. A community-wide standard was a discussion topic at several working group meetings during DASADA PI meetings. This was to be finalized in the Phase II externalized infrastructure that was never funded.

It is up to individual probe technologies to determine whether multiple probe configurations (defined in the same probe technology) can be installed on a single system without interfering with each other. Those that can't should issue an exception when more than one Install is issued for an execution instance.

Activate(Probe-Configuration-Name, Host, System-Spec)

The already installed Probe-Configuration-Name on System on Host is activated so that it senses behavior in the running system. In this activated state, the probes may issue Sense events for some subset of the behavior they observe. If an Activation event, as well as an Install event, is received before the named System is running, then Probe-Configuration-Name is both incorporated into that System when it is started on Host and immediately activated.

Sensed(Probe-Configuration-Name, Host, SystemExID, Event-Name, Event-Type, Data1 ... DataN)

Probes in the activated state may issue Sensed events that identify some subset of the behavior they observe. The Event-Name identifies the type of behavior observed and Data1 through DataN are values detailing that type of behavior. Event-Type is either Start, End, or Point which specify respectively the start of an event interval, the end of an event interval, or the occurrence of a point event (i.e. an event with no duration). Probe-Configuration-Name specifies the probe configuration that generated the event and SystemExID specifies the particular execution instance running on Host on which that probe configuration was installed.

Query-Sensed(Probe-Configuration-Name, Host)

Requests a list of all of the Event-Names that the named Probe-Configuration-Name can generate while it is activated. This request is answered through a Generate-Sensed event. As this is a static property of the probe configuration itself, no execution instance need be specified.

Generate-Sensed(Probe-Configuration-Name, Host, Event-Name1 ... Event-NameN)

This event is issued in response to a Query-Sensed event for the list of all Event-Names that the named Probe-Configuration-Name can generate while it is activated. This list of Event-Names is Event-Name1 through Event-NameN.

Focus(Probe-Configuration-Name, Host, SystemExID, Parameter1 ... ParameterN)

Parameter1 through ParameterN are passed to the already installed Probe-Configuration-Name on execution instance SystemExID on Host to "focus" its sensors as specified by the parameters. How it interprets these parameters is entirely probe configuration specific.

Deactivate(Probe-Configuration-Name, Host, System)

The already installed Probe-Configuration-Name on execution instance SystemExID on Host is deactivated so that it stops sensing behavior in that execution instance. In this deactivated state, the probes may not issue any Sensed events.

Uninstall(Probe-Configuration-Name, Host, SystemExID)

The Probe-Configuration-Name on Host is removed from execution instance SystemExID or its environment. It cannot be uninstalled if it is currently active (i.e. it has been activated more recently than deactivated) It can no longer be Activated or Deactivated on that execution instance before it is again Installed on that execution instance.

Undeploy(Probe-Configuration-Name, Host)

The named Probe-Configuration-Name is no longer a defined probe configuration on the named Host and can no longer be referred to by that name on the named Host. However, installed instances of the Probe-Configuration-Name are unaffected by the Undeploy.

Adaptor

A probe technology may not be able to directly issue and respond to the events defining the Probe Run-Time Infrastructure APIs. It therefore has an Adaptor that couples the probes defined in that technology to the Probe Run-Time Infrastructure.

The Adaptor on a machine receives all the events passing through the Probe Run-Time Infrastructure for the probe configurations on that machine. It converts those events into calls and operations in that probe technology. As an example, it may translate an Active event into the setting of a global variable that acts as a guard for probes already in place.

The Adaptor will also receive any "outputs" generated by these probes and convert them into events to be passed through the Probe Run-Time Infrastructure.

Community Support for Probe Infrastructure

(See Run-Time-Infrastructure Working Group under the DASADA Community Relationships section.)

Technical Result: An En-gauging Architecture Framework

The framework we designed and implemented for monitoring running systems was demonstrated on a showcase program designed and implemented by Teknowledge, a program called “Safe Email” [Balzer ‘01b] that protects users from malicious attempts to install worms or viruses into the user’s workspace via attachments or other sources. Although our original framework was not developed as a wholly externalized adaptation of the running email program, it is best described in the terms of the framework proposed for the DASADA Phase II follow-on.

Hence, in what follows, the essence of that proposal for the externalized framework is described before proceeding to describe the actual program design for monitoring safe email.

As the DASADA program unfolded, one of our realizations was that Gauge technology is not only applicable for maintaining and improving QoS concerns, but also for producing *self-healing* effects. Hence, along with others from DARPA’s DASADA program we proposed an execution infrastructure for so-called self-healing, self-adaptive systems – systems that maintain a particular level of healthiness or quality of service (QoS). This externalized infrastructure does not entail any modification of the target system whose health is to be maintained. It is driven by a reflective model of the target system’s operation to determine what aspects can be changed to effect repair.

The infrastructure can be thought of as a specific architectural “harness” to facilitate dynamic system reconfiguration, as required for these “self-healing” or “self-adaptive.” Figure 1. An Externalized Infrastructure for Self-Healing Systems illustrates an architectural diagram of an externalized infrastructure that can be used to monitor, interpret, analyze, and reconfigure running systems. Essentially a layer of probes is installed into a system just before it starts to run, probes that report significant data on a probe event-bus (described in the previous Technical Result section). Gauges translate and interpret this data with respect to models that abstract from the implementation, often using an architectural model of the target system to locate logical events.

The information from these gauges is transmitted onto the gauge bus where other gauges can react and control decisions can be made. A layer of "effectors" is then invoked to effect changes in the target system, either by adapting existing components – perhaps by tweaking parameters – or by reconfiguring the system itself.

This infrastructure was proposed by several members of DARPA's DASADA community [Garlan and Schmerl] [Valleto and Kaiser] [Wile '02] where it was only partially developed before funding was dropped; however, the probe and gauge interaction protocols were standardized [Balzer '01a] [Garlan, Schmerl, and Chang] based on Siena event-broadcasting middleware [Carzania et al].

Architectural Modeling

The Target Architectural Model (modeling the "cloud" in Figure 1) plays a key role in the infrastructure. The architectural model essentially establishes the structural vocabulary; each layer can rely on the model for understanding its role in the system and the information it is responsible for. To see how events in the physical architecture map to events in the logical, target architecture model consider the following scenarios, both ensuing after probes and gauges have been placed by the control layer:

Static Architecture Scenario

- Probes emit Implementation-Level Events (ILEs) like "process D006 opened file 'C:\Program Files\log.txt' for write" or "process E001 used 2021."
- Gauges provide interpretations of these events by first determining what logical architectural entities are being referred to – here, perhaps a logical application, WinZIP (D006), and another logical application, MS PowerPoint (E001), for example. This mapping from implementation terms (process ids) to logical architectural components must be established in the architectural model by the processes that originally set up the system and probes. The gauges additionally interpret implicit information from the probes; for example, perhaps 2021 means port 2021.
- The gauges are then "read" by the control layer to see if any action should be taken. For example, assume that the ILE for E001 is interpreted as "MS PowerPoint (E001) is attempting to access port 2021." Furthermore, assume that the control layer has knowledge of good, suspicious, and bad events. For example, it is known that "access to ports 1000-3000 is suspicious," e.g., because normal application operation does not require such access. In such a case, the control layer may decide to ask the user of the application to authorize or deny access to port 2021. The control layer may then communicate the user response to authorize or deny the access to the effector layer through an adaptation event (AE).

- Then the effector layer will use the architectural model to determine that process E001 needs to be adapted – requiring the inverse translation from before, now from logical architecture to physical architecture – and determine what implementation-level response corresponds to authorize or deny events, e.g., raise an implementation-level "port access failed exception" in the latter case.

Notice that nothing about the architecture itself changed during this scenario; no modules or connections were created or destroyed. Moreover, the repair was affected by a simple parameter change to a running module; no new resources were brought to bear. A similar scenario might require dynamic target architecture changes:

Dynamic Architecture Scenario

- Probes emit architecturally significant implementation-level events, such as "process D006 spawned new process F008 of type MS Word."
- Gauges interpret these events and modify the corresponding physical and logical architectural models. Here, perhaps, because F008 was spawned by D006, the system knows that there must be a new logical application, MS Word (F008) now and that the logical application WinZIP (D006) is the creator (parent) of MS Word (F008). We call this process *identification* of physical models with pre-defined, logical architecture models. That is, with dynamic architectures, the whole range of possible architectures is pre-specified in a covering architecture [10,13]; those elements of the architecture that have been identified with the physical architecture are kept track of. Hence, at any given time, only the identified modules and connectors constitute the actual logical architecture.
- Imagine that some time later an event similar to the one above, "MS Word (F011) is attempting to access port 2021" is transmitted by the probes and reported by the gauges. The control layer at this point could issue a user request to authorize/deny this attempt or it could change the system's running architecture by issuing a *reconfiguration* event to the effector layer. This time perhaps the command issued would be to "replace the MS Word process (F011) with another physical application (e.g., MS WordPad, which can read MS Word documents but is less subject to exploitation by viruses).
- The effector layer again has to map the logical MS Word component into the physical process F011 and it also has to understand how to remove that component and substitute a new one of type MS WordPad, a rather tricky activity in any event.

So there are two separable dynamic architecture activities here: modeling the dynamic architecture as it evolves and reconfiguring the architecture via the control layer.

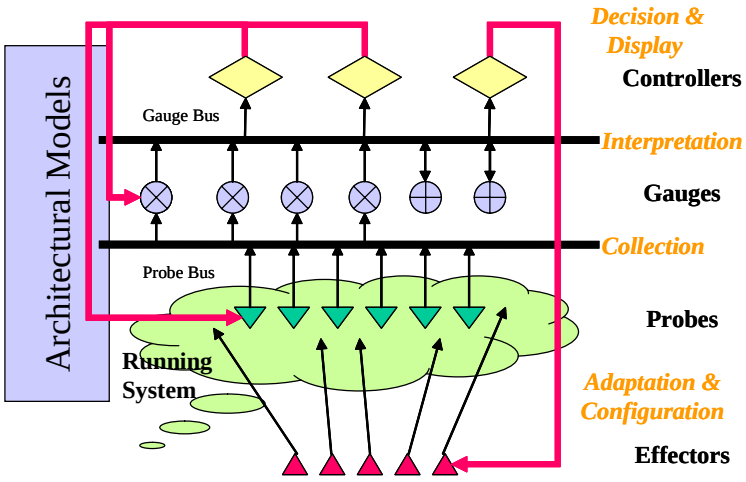

Figure 1. An Externalized Infrastructure for Self-Healing Systems

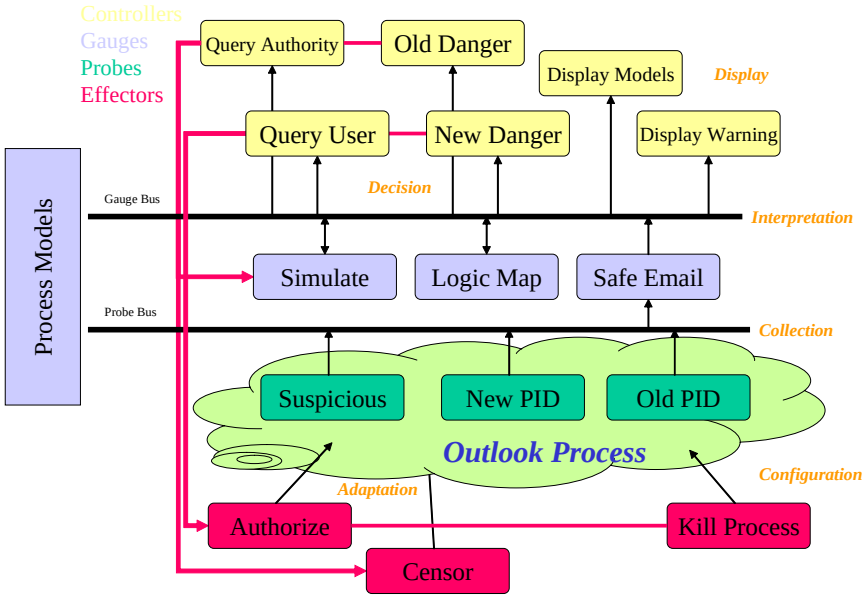

Figure 2. Idealized Safe Email Infrastructure Architecture

An Example: Safe EMail

The following is a reformulation into the self-healing harness of the running program the En-gauging Architectures project developed and demonstrated. In effect our "Safe Email" program [Balzer '01b] itself constitutes a self-healing harness for repairing the errant behavior of processes spawned by email attachments.

Figure 2 depicts the instantiation of the Safe Email self-healing infrastructure (Figure 1. An Externalized Infrastructure for Self-Healing Systems) for the email application Microsoft Outlook. Its purpose is to detect and prevent malicious behavior caused by viruses received through email. Email viruses are either embedded in the email itself to exploit security weaknesses in MS Outlook (e.g., macro viruses) or they are unleashed by attachments opened by unsuspecting users.

To counter the security threat posed by viruses, three kinds of probes are required, probes for observing: 1) the occurrence of events that could be considered "suspicious," 2) the creation of new processes (New PID), and 3) the destruction of existing processes (Old PID). All three probes are based on wrapper technology, where calls to PC Windows-based platforms' Dynamic Link Libraries are intercepted and our code is invoked before (conditionally) invoking the original code [Balzer and Goldman], reporting suspicious activity via the probe bus [Balzer '01a]

If a virus exploits a weakness in MS Outlook, then it will engage in suspicious activities that are observed through the first type of probe. The other probes maintain the target architectural model to coordinate faults with spawned processes. When attachments are opened, new processes are created to view/execute these attachments (e.g., MS Word or a Web Browser). Therefore, if a virus is embedded in an attachment, the new process, and not MS Outlook, will then engage in suspicious activities. Observing "suspicious" activities is thus extended to processes spawned by MS Outlook (New PID) until they are destroyed (Old PID).

The "Safe Email" gauge acts as a mediator to collect and translate probe information broadcast on the probe bus. It may combine multiple implementation-level events to produce architecture-level events that abstract from the implementation. It may also translate observed information with the help of the "Logic Map" gauge to interpret how implementation-level data relates to target architectural elements and to record the creation hierarchy of applications. MS Outlook sits at the top of this hierarchy. When it spawns a process by opening an attachment, a "child application" is created to represent the new process along with its type and id. Since a spawned process may spawn yet other processes, the target architecture model supports a tree hierarchy of "parent" applications and their "children."

Gauges cannot judge whether suspicious activities, caused by MS Outlook or any of its child processes, are truly malicious or not. If a suspicious event is observed the first time ("New

Danger") then the control layer displays a warning message ("Display Warning") to let the user decide about the maliciousness of the event ("Query User"). The user may allow the activity, deny it, or kill the application. The user may also reconfigure the control layer to ignore similar events ("Old Danger") in the future (equivalent to an automatic allow).

Probes may reside in different processes and on different machines, so the infrastructure can be used to monitor multiple email users. A so-called "Authority" is given access to a GUI showing the target architectures with processes decorated by border colors indicating how well behaved processes are with respect to producing suspicious activities that the users deny, e.g., red for malicious.

In addition, the architectural elements are simulated in the "Simulate" gauge to determine the level of trust of individual applications, based on users' responses to warnings of suspicious activities and to "guilt assessment" imposed on parents of misbehaving child processes [Egyed and Wile]. This information is also visualized for each process.

The authority is allowed to determine that specific processes are misbehaving for enough users that subsequent attempts to invoke the suspicious actions should automatically be denied. "Query Authority" uses MailIDs to ensure that previously denied events are denied automatically again if they originated from the same email, e.g., even for different users.

The layer of effectors is invoked to effect changes in the running system. Effectors may "authorize" or "censor" (deny) suspicious events, or may even kill processes.

Harness Problems

Although we noticed problems in applying the infrastructure to the Safe Email example, we think this approach is a feasible way to decouple self-healing aspects of a target system from its functionality. Problems with externalization arise when there is a coupling between an effector that corrects a problem and a sensor that detects it, e.g., a sensor detects danger and suspends itself, awaiting a decision about how to proceed. The effector that allows it to proceed is strongly coupled, something that cannot be indicated with the infrastructure as it stands. Similar problems concern how to model the user and administrator.

Nonetheless, we feel the future of this infrastructure will best be to serve as "a template" for imposing self-healing systems on applications, as suggested by Jeff Magee [Crane et al]. In fact, in the future we intend to design an *architectural style* that is consistent with the infrastructure that allows refined descriptions of the relationships of the sensors, gauges, controllers, and effectors. The choice of which architecture description method to use – infrastructure or style – will then depend on the volatility of the infrastructure itself.

Design Result: an Architecture Meta-Language

Dynamic evolution concerns arise with considerable variation in time scale. One may wish to constrain how a system can evolve over its development lifecycle. Laws such as Minsky [Minsky] proposes or constraints as in Monroe's Armani system [Monroe] address such evolution concerns. Another approach to such concerns involves limiting systems' construction primitives to those from appropriate styles, such as in Wright [Allen] and UniCon [Shaw], or embodied by the choice to use C2 [Oreizy, Medvidovic, and Taylor]. One may wish to constrain what implementations are appropriate; concerns for interface compatibility such as evidenced in SADL [Moriconi et al] are then germane. And finally, one may want to constrain the ability of the

architecture to be modified as it is running; languages such as Rapide [Luckham et al] and Darwin [Magee and Kramer] emphasize these issues.

The language AML was designed under this contract to be used to specify the semantics of architecture description languages (ADLs). AML attempts to provide specification constructs that can be used to express all of the constraints mentioned above without committing to which time scale will be used to enforce them. It is a very primitive language, having declarations for only three constructs: elements, kinds, and relationships. Each of these constructs may be constrained via predicates in temporal logic. The essence of AML is the ability to specify structure and to constrain the dynamic evolution of that structure.

The full description of AML is presented in an attachment, but to summarize here: software architectures are to be represented as a set of elements among which distinguished topological relationships are carefully described and constrained. In fact, these relationships will be time varying, or event-based. This mostly affects the logical framework needed to reason about them. In effect, AML is designed to facilitate specification of these concepts only: elements, topological relationships, domain-specific relationships on the elements, and temporal constraints, along with facilities for organizing and describing these concepts more concisely.

AML semantics require several different validation and verification activities on the part of its users and/or support tools. In addition to formal proof obligations, one must:

- Identify the elements of the model with items in the artifact. (It is assumed that no two differently named items be identified with the same artifact.)
- Ensure that these identified elements satisfy appropriate topological relationships, again in the artifact itself. Specifically, if an element is identified, this usually requires that other topologically related elements be identified as well.
- Ensure that certain closure properties hold in the artifact. This involves establishing that e.g. all of the parts, and only the parts are accounted for that have the part relationships.
- Establish the non-topological properties. This is a purely domain-specific activity, and is actually the major source of leverage of ADLs.

The semantics of AML require only a very small part of the predicate calculus along with some elementary set theory. It is our intention that AML be adapted to different logics for different analysis purposes. A good starting ground for such a logic includes simple temporal predicates, *sometimes* and *always*, conventional quantifiers with typed variables, *exists* and *all*, as well as the connectives for *implies*, *equivalence*, *exclusive or*, *inclusive or*, and *and*. The connectives connect potentially negated comparison relations or propositions. Infix versions of the standard comparison relationships should also be included, e.g., <, >, >=, =<, <> and =.

The structural building block in AML is the relationship. Over the past 25 years in the Software Sciences Division at ISI we have built many languages based on “relational abstraction” wherein we model all data access as manipulations of abstract relationships [Feather]. Some of the conventions adopted there are brought over into AML. In AML the relationship declaration is used to describe topological relationships among elements and domain relationships between elements and other external types, such as integers or strings, or even modules in a programming language, for example.

Naturally, the whole purpose of AML is to specify architecture *constraints*. Elements may be constrained in two ways: through constraints that are assumed to be true of the element and through constraints that should be able to be logically derived as holding. The constraints that are assumed to hold, including the unique identification axiom (that all elements have distinct ids), must be validated in the artifact being modeled. Generally, topological relations will be

introduced along with assumptions they require, as with the single parent assumption for the *has-part* relationship. In addition to topological relationships, those that relate different elements of the architecture, other interesting relationships include those between architecture elements and other models. In fact, almost all of the power of an ADL stems from its ability to model some aspect of the application of the architecture that can be analyzed and reasoned about a priori. As with topological relations, assumptions and definitions of other relationships may be introduced explicitly in the context of particular elements by indenting appropriately. There are several benefits to introducing them this way: (1) the scope rules apply, so elements can be referenced without giving their full "lineage" (using dot notation); (2) locality will tend to correlate with relevance (although one can reference elements freely outside the scope of the assumption); and (3) conditional identification of elements can be implicit.

In AML *kinds* play the role of both architecture types and architectural styles, introducing and restricting new element types. The word was chosen for its lack of associations in modern programming or specification languages. A kind declaration looks like an element declaration, but the closure properties are extended somewhat to provide flexibility when the instances of kinds are specified.

In the introduction to this section, we mentioned that one may wish to constrain how a system can evolve in very many parts of its development lifecycle. Laws such as Minsky proposes [Minsky] or constraints as in Monroe's Armani system [Monroe] address such evolution concerns during system specification. Another approach to such concerns involves limiting systems' construction primitives to those from appropriate styles, such as in Wright [Allen] and UniCon [Shaw], or embodied by the choice to use C2 [Taylor]. One may wish to constrain what implementations are appropriate; concerns for interface compatibility such as evidenced in Structural Architecture Description Languages (SADL) [Moriconi et al] are then germane. And finally, one may want to constrain the ability of the architecture to be modified as it is running; languages such as Rapide [Luckham] and Darwin [Magee] emphasize these issues. Using AML's specification constructs allows one to express all of these constraints without committing to which time scale will be used to enforce them. Each of the above approaches should map readily into AML. Naturally, different logical systems may be necessary. That is consistent with the philosophy of AML. If more restrictive constraints require more "reflective" capabilities of an ADL than are present here, one should seek to regularize them and introduce them into AML.

The abstractness of the notion of topological and domain relationships may mask the general usefulness of this approach. For example, topological relationships include the obvious part-of, port-of, component-of, connector-of, attached-to, etc., but also may be used to represent more detail, such as refines-to and binds-to, to represent architecture at a finer level of detail. Domain relationships will include somewhat generic ones such as implemented-by and throughput, but the leverage of ADLs is in the richness of the domain relationships for describing events and their interactions. AML simply provides a foundation for relating them to the topological elements of an artifact.

Normally, the existence of a formal "semantics" for a language is simply a confidence-building device for its designers. An interesting consequence of AML semantics is that they can actually be useful in realistic settings. If the topmost element of an architectural description is presumed to be identified, there will generally be several other elements that must be identified as well. If these are furthermore assumed to hold, there will still be a residue of assumptions about the identification of optional elements and replicated elements that cannot be assumed.

This residue can be manipulated and used in testing the running architecture for conformance, for example, by instrumenting it with probes that detect the identification and perhaps subsequent non-identification of elements. A "shadow architecture" specification – the residue – can then be

monitored for compliance. The Flea system [Naryanaswamy] has been used for such purposes, although the residue was concocted in an ad hoc fashion. In fact, we fleshed this idea out in more detail in a paper presented at the Requirements Engineering Conference in 2000 [Wile '01a] (see attachment). Moreover, these ideas were incorporated into the Acme language specification and reported in [Wile '01c] (see attachment). For more information AML itself is described in more detail in [Wile '01b] (see attachment).

Design Result: Synthesis of Dynamic Architecture Event Languages

A variety of architectural modeling tools have been designed and implemented under DARPA's EDCS and DASADA programs with the goal to facilitate dynamic system reconfiguration. In Figure 1 above, we illustrated an architectural diagram of an externalized infrastructure that can be used to monitor, interpret, analyze, and reconfigure running systems. This infrastructure was only available piecemeal, but several parts were developed to the extent that it was time to standardize some of the interaction protocols to provide leverage to the community developing self-aware, self-healing systems. Notice the importance of the architectural model to this diagram.

Here we focus on the issues involved in designing a protocol for architecturally significant event (ASE) description, that is, events that indicate construction or destruction of architectural components themselves. Recall the static and dynamic scenarios above where ASEs were described.

There are three separate protocols in use for describing architecturally significant events (ASEs)[2] in the DASADA community presently. The xArch/xAcme [Schmerl and Gross] design provides events expressed in XML to describe the changes to connectors, components, attachments, etc. This formalism is based on a tool support mechanism for Acme [Garlan, Monroe, Wile], called AcmeLib [Schmerl]. The formalism actually has extensions to deal with CMU's proposed gauge infrastructure [Garlan, Schmerl, and Chang], but they have been eliminated here as representing a somewhat ad hoc, interim solution.

A second protocol also uses XML to describe sets of architecture changes to an architecture expressed using the xADL architecture description language [van der Hoek and Dashofy]. This so-called "Diff" formalism is primarily used to describe differences between successive versions of architectures [van der Westhuizen and van der Hoek] for version control purposes.

Finally, our work with the PowerPoint Design Editor [Goldman and Balzer] embodies an ASE description language. This was originally described as COM events but was recently converted to call-backs from a PowerPoint add-in.

Each of these ASE languages is characterized by the abstract syntactic structure of the events allowed. The xAcme/xARCH[3] schema comprises create, delete, attach, detach and changeProperty events. Create takes a sequence of additions to be performed in a particular context, presumably the system being modeled. These additions include components, connectors, interfaces, sub-architectures, and bindings, as well as a second-order element that is a property

[2] Here there is no way to distinguish "implementation level" events from more abstract reporting, so the "IL" has been dropped for the rest of the paper.

[3] These two XML dialects differ in that Acme's allows explicit properties on the elements where xARCH simply allows normal XML to carry the properties as extra fields.

description. Deletion, on the other hand, just removes the element pointed to. Attach and detach both take sequences of role-port pairs as arguments. And changeProperty changes the property from whatever value it has to the argument value.

The xADL "Diff" interface is slightly simpler. Diff starts out with a whole sequence of Add and Remove events. Things that can be added are: components, connectors, links, groups, and type descriptions for the first three of these. Deletion is by reference to the architecture instance identifier. Although typing is not specified at this level in the other two protocols, the simplicity of the remainder here arises in part because of the lack of property definitions but primarily because one must understand the substructure of the architectural elements to completely construct the artifacts. For example, what a link links is implicit in the definition for link; it is not present in the Diff interface itself. In both other protocols such elements are explicit. Similarly, grouped elements are found within the group, rather than within the Diff interface. The implications of this will be discussed below.

The Design Editor appears to be somewhat more complex (and primitive) than the other two protocols, in that no grouping constructs analogous to XMLs sequences are used except in the new Property event. (The Design Editor allows multiple-valued properties; Acme would represent these as set-valued properties.) Sequences of these events between begin- and end-transaction events define the grouping instead. There is a slight disconnect in the nomenclature of the Design Editor that makes comparison a bit difficult. Here, both components and connectors are referred to as "shapes;" their classification as to either must either be derived from the type structure specification (extraneous to the protocol), or from their roles in the OutboundConnect and InboundConnect events. There is a lot of explicit information given to the (new) Shape event. In particular, the shape's ID (guaranteed to be unique over all components and connectors in any given architecture[4]), its type, and information relating to whether it is itself a sub-architecture (the abstract) parameter and/or its membership in one. This information is also explicitly present in the xAcme/xARCH protocol, but is again implicit in the xADL one.

The Design Editor restricts connectors to exactly two roles - an inbound and an outbound one - and components to ports represented as integers. (These ports can be named but one cannot derive that information from what is passed through the protocol.) All of the elements of this protocol are represented as integers (Longs) except for the names and values of properties. The values are actually strings or integers, but that fact cannot be determined from the weak typing in the protocol.

These apparently simple, apparently similar protocols are different enough to illustrate several issues that must be addressed in any synthesis of them into a commonly acceptable ASE. In particular the following stand out:

- Goals for the protocol. What belongs in the protocol?
- Nomenclature issues. Can we agree on a nomenclature or is a Rosetta Stone appropriate?
- How many different representations of the events are needed? Is XML sufficient?
- How rich should the event language be? For example, the union of the three languages would be a possible language proposal (absurd, but possible). Or should a core facility be extendable, and how?
- What transaction model should be used? Explicit begin-end, nested transactions, set of changes, sequence of changes, higher-level operators encapsulating sequences - such as "change" for "remove and then add."

[4] Technically, any PowerPoint presentation.

- How much extraneous information - referred to as implicit information above - is appropriate to understand the artifact that the ASEs can be used to construct? How does one identify an architectural element uniquely?

To approach the first issue above, we employed the use of an abstraction I first used when describing the DARPA EDCS program's legacy to DASADA principal investigators [Wile '00] (see attachment). In particular, there seem to be four levels of specification of architecture present to varying degrees in ADLs and their support technologies:

- Core (Syntactic)
- Constrained (Type checked)
- Completed (Analyzed)
- Reflective (2nd Order Representation)

These levels correspond to successively deeper levels of commitment to representations and tools. In the protocols studied, the issue was to come up with a representation of events that construct only the core architectural elements - connectors, components, attachments, groups and their syntactic substructure and properties. Of course, there are some elements of the constrained level as well, but the type checking activity itself occurs outside of these protocols. The completed layer refers to analyses that can occur only after one claims to have specified all of the information needed to reason effectively. For example, one cannot hope to reason about performance if an arbitrary component may still be added to the architecture, perhaps connected between two existing components. Aspects of reflective capabilities are already in the protocols described: xAcme's newProperty and xADL's element-types are of this nature.

To consider the issues above in more depth, it might also help to constrain the protocols to suit the DASADA infrastructure depicted in Figure 1. This would be a secondary goal, if a more general solution is satisfactory, but e.g. one might be able to assume that the architecture was *type-correct* by construction of the probes that monitor the system. One could assume that some aspects of *analyzed* were true by construction as well. For example, security properties may be guaranteed by a wrapper technology despite component volatility.

The following were points of discussion toward resolving the issues above during the Self-Healing Systems Conference in Charleston, SC [Wile '02], although the time allotted for discussion was too short to come to any firm conclusions. Again, funding for DASADA Phase II was intended to resolve these issues.

Proposal Goals. It would seem wise to look ahead to the constrained and completed levels of representation to see how to handle some of the issues that arise already at the low level, even if we do not intend to support them. However, I actually believe a modicum of support for both levels can be provided as well.

Nomenclature Issues and Different Representations. I believe agreement at the level of a common API should be sought. The Probe Infrastructure Proposal of DASADA [1] was developed as an API for events that manage sets of probes called configurations. The important aspect of that proposal here is that various underlying technologies can be used to implement the substrate supporting the API.

I can imagine doing the same here, using what the probe infrastructure called "adapters" to allow interoperation between various actual dialects used to implement the API. In any case, a shared, abstract syntax-based API design seems appropriate. That way we can call an attachment a link or a beginConnect with impunity at the substrate level, but talk uniformly with one another at the abstract level.

Language Richness. I believe the adapters could be used to ameliorate this as well, but only to the extent that we disagree! The point is that the higher the level we can agree to use in the common API, the better off we are for enhancing sharing and functionality. I personally believe that named type-equivalence should be built into the protocol, but I expect some resistance from the xAcme/xARCH camp.

I would also propose to extend the language to facilitate some of the reasoning used in Armani [Monroe] and Dynamic Acme [Wile '01c], where architecture "closedness" is important: an architecture description characterizes all possible architecture instances - the "proto-architecture" - while running instances inform of the choices made within that space. Strong analogies between the Probe Infrastructure API for "installing" configurations versus "activating" them exist in almost all dynamic architecture descriptions. In order to give some support for the constrained and completed layers, it would seem important to carry that distinction over into the ASE protocols as well. A simple starting proposal here is to add an "activated" or "identified" event to the protocol that can be used to indicate the dynamic status of architecture elements.

Transaction Model. That proposal almost requires that a transaction model be incorporated (or at least, permitted). It may even argue for a two phase transaction, one in which the proto-architecture is instantiated and the other for dynamic activations. It should be noted that indeed this idea is allowing the dynamic introduction of reflective knowledge in the system, something shown to be important to a system as simple as the Probe Configuration Manager itself.

The API could allow transactions as well as grouping constructs and perhaps even atomic sequences, as represented by the change operator, i.e. allow the union of these facilities found in the current protocols instead of just choosing one. I believe that the API design might not need to be implemented in its entirety by every adapter; we could register those features an adapter supports and not utilize the adapter when it proves to be insufficient, for example. However, this is an issue for discussion as well.

Extraneous Information. I believe the model of the dynamic architecture created by the ASEs should be uniquely determinable up to isomorphism of textual identifiers and strings. The xAcme/xARCH and the Design Editor protocols are closer to this than the xADL, but some work is needed to establish this with any of them. Architectural element identity will probably be a tricky issue. Hopefully, it will not require anything as heavy-handed as the use of urls, for example.

Key Personnel

In addition to the project's two Principal Investigators, Robert M. Balzer and David S. Wile, Alexander Egyed participated in the project as developer of the Simulation tool used in the dynamic architecture self-healing gauge. Neil Goldman assisted in the development of the Probe wrapper tools that implemented the Probe Infrastructure Specification.

Key Trips and Presentations

Both David S. Wile and Robert M. Balzer attended all DASADA PI meetings and the so-called "Demo Days" meetings, where results were presented to the wider Washington, D.C. community.

DASADA Kick-off meeting. Santa Fe. 7/11/00-7/14/00 David Wile gave a presentation entitled: "The EDCS Architecture Legacy" (see attachment).

PI Meeting. Monterey 1/31/01-2/2/01. Robert Balzer presented the probe infrastructure proposal.

Demo Days. Baltimore. 6/4/01-6/8/01. Robert Balzer, Alexander Egyed, and David Wile demonstrated an early version of the Safe Email program en-gaugement using the PowerPoint Design Editor on a Probe Architecture Style representation (see attachment).

PI assembly. Brisbane. 12/15/01. David Wile and Robert Balzer attended in conjunction with the Working Conference on Complex and Dynamic Systems Architecture (see below).

Infrastructure meeting. Stanford Research Institute. 3/18/02. David Wile attended and presented an example using Dynamic Acme (from the working conference).

Demo Days. Baltimore. 7/1/02-7/4/02. David Wile and Robert Balzer attended and presented the demo entitled: Architectural Gauges (see the DASADA Brochure and other 2002 PowerPoint presentations in the attachments).

Phase II planning meeting. Arlington. 7/23/02-7/24/02. David Wile attended and supported the consolidated proposal (see Figure 1 ff.).

David Wile had a paper entitled "Modeling Architecture Description Languages using AML" accepted to the Journal of Automated Software Engineering. 8: 2001. 63-88 [Wile '01b] (see attachment), with the following abstract:

> The language AML was designed to specify the semantics of architecture description languages, ADLs, especially ADLs describing architectures wherein the architecture itself evolves over time.[5] Dynamic evolution concerns arise with considerable variation in time scale. One may constrain how a system may evolve by monitoring its development lifecycle. Another approach to such concerns involves limiting systems' construction primitives to those from appropriate styles. One may wish to constrain what implementations are appropriate; concerns for interface compatibility are then germane. And finally, one may want to constrain the ability of the architecture to be modified as it is running. AML attempts to circumscribe architectures in such a way that one may express all of these constraints without committing to which time scale will be used to enforce them. Example AML specifications of the C2 style and Acme are presented.

Alexander Egyed attended the Working IEEE/IFIP Conference on Software Architecture (8/01, Amsterdam) and presented a paper entitled "Statechart Simulator for Modeling Architecture Dynamics" [Egyed and Wile] (see attachment) with the following abstract:

> Software development is a constant endeavor to optimize qualities like performance and robustness while ensuring functional correctness. Architecture Description Languages (ADLs) form a foundation for modeling and analyzing functional and non-functional properties of software systems, but, short of programming, only the simulation of those models can ensure certain desired qualities and functionalities.
>
> This paper presents an adaptation to statechart simulation, as pioneered by David Harel. This extension supports architectural dynamism – the creation, replacement, and destruction of components. We distinguish between design-time dynamism, where system dynamics are statically proscribed (e.g., creation of a predefined component class in response to a trigger), and run-time dynamism, where the system is modified while it is running (e.g., replacement

[5] This work was sponsored by the Defense Advanced Research Projects Agency under contract nos. MDA903-87-C-0641, DABT63-91-K-0006, and F30602-96-2-0224.

of a faulty component without shutting down the system). Our enhanced simulation language, with over 100 commands, is tool-supported.

David Wile attended the Requirements Engineering (8/27/01-8/31/01, Toronto) conference and presented the paper entitled: "Residual Requirements and Architectural Residues" [Wile '01a] (see attachment) who's abstract is:

> Monitoring running systems is a useful technique available to requirements engineers, to ensure that systems meet their requirements and in some cases to ensure that they obey the assumptions under which they were created. This report studies relationships between the original requirements and the monitoring infrastructure. Here we postulate that the monitored requirements are in fact just compilations of original requirements, called "residual" requirements.
>
> Dynamic architectural models have become important tools for expressing requirements on modern distributed systems. Monitoring residual requirements will be seen to involve "architectural residues," skeletal run-time images of the original logical architecture. An example sales support system is used to illustrate the issues involved, employing modest extensions to the Acme architecture description language to reason about architectural dynamism.

David Wile attended the Working Conference on Complex and Dynamic Systems Architecture (12/9/01 – 12/11/01, Brisbane, Australia) and presented a paper entitled "Using Dynamic Acme" [Wile '01c] (see attachment) with the following abstract:

> Dynamic architectural models have become important tools for expressing requirements on modern distributed systems. I previously identified "architectural residues," skeletal run-time images of the original logical architecture, as important reflective models to be maintained in monitoring that running systems meet dynamic requirements. Several years ago I proposed modest extensions to the Acme architecture description language to express aspects of, and reason about, architectural dynamism. Herein these extensions are made more precise; an example illustrates role use in inserting probes and gauges into systems.

David Wile attended the Workshop on Self-healing Systems (WOSS, 11/18/02- 11/19/02, Charleston, S.C.), and presented the paper [Wile '02] (see attachment) with the following abstract:

> Self-healing systems generally require reflective models of their own operation to determine what aspects of themselves they can change to effect repair. Architecture models are examples of rather simple models to which health information can be attached and reasoned about, e.g. attaching system state to a process or tracking events across connectors. These models are especially useful when the architecture of the system varies while the system is running, in so-called "dynamic architectures."
>
> DARPA's DASADA program is developing an architecture-based infrastructure for self-healing, self-adapting systems. Herein several protocols for dynamic architecture change notification from that program are examined in search of a community standard for such a protocol. Desirable properties of such protocols are suggested based in part on how much constraint checking will be used to proscribe dynamic architecture building activity. Points for discussion are raised.

David Wile attended the Conference on Automated Software Engineering (ASE, 10/6/03-10/10-03, Montreal) and presented the short paper entitled "Calculating Requirements: an Approach Based on Architecture Style" [Wile '03] (see attachment) with the following abstract:

> Engineers wield various "calculi" to help determine solutions to their problems, calculation tools varying in power from tensile strength tables to the differential calculus. Software engineers sometimes use domain-specific languages that provide calculi for their tasks, for domains as varied as music composition, control system design, and parsing. The idea here is to explore what it would mean to provide calculi for requirements engineers to aid them in solving their problems. An approach to designing such calculi is presented that is based on the use of architecture styles. An example calculus based on the model / view / controller architecture style is sketched to demonstrate that rules for manipulating these architectural elements can aid in requirements "calculation."

David Wile and Alexander Egyed had a short paper accepted to the Working IEEE/IFIP Conference on Software Architecture (to be held 6/13/04-6/15/04, Oslo) entitled "An Externalized Infrastructure for Self-Healing Systems" [Wile and Egyed] (see attachment) with the following abstract:

> Software architecture descriptions can play a wide variety of roles in the software lifecycle, from requirements specification, to logical design, to implementation architectures. In addition, execution architectures can be used both to constrain and enhance the functionality of running systems, e.g. security architectures and debugging architectures. Along with others from DARPA's DASADA program we proposed an execution infrastructure for so-called self-healing, self-adaptive systems – systems that maintain a particular level of healthiness or quality of service (QoS). This externalized infrastructure does not entail any modification of the target system – whose health is to be maintained. It is driven by a reflective model of the target system's operation to determine what aspects can be changed to effect repair. Herein we present that infrastructure along with an example implemented in accord with it.

In addition, both David S. Wile and Robert M. Balzer attended several conferences, workshops, and IFIP working group meetings (WG2.1, WG2.2 and WG2.9) where the DASADA efforts were reported and discussed enthusiastically.

DASADA Community Relationships

The DASADA community was considerably more aware of what others were doing in related areas than most DARPA programs that these authors have been a part of. Several ad hoc working groups were formed to establish community consensus in areas of common concern. The topics of these included: run-time infrastructure and the common extensible design notations groups as well as others with which we were less involved (such as effectors).

Run-Time-Infrastructure Working Group (Probes)

Bob Balzer (Teknowledge) was the chairman of the committee to develop the probe infrastructure specification and was the author of the strawman proposal that was ultimately adopted by the group. Several members developed tools that conformed to it, including Teknowledge Corp. (Balzer, Goldman), Colombia University (Kaiser), and OBJ (David Wells). They decided to use the Siena event bus carrier mechanism along with a primitive event mechanism, awaiting a design by the Common Extensible Design Notations committee. Teknowledge later substituted a secure transport mechanism that was consistent with the probe infrastructure for use in its demonstrations. Later on, this committee began to formalize the notion of a "gauge bus" in parallel with the probe bus. (See Figure 1, where both of these are evident.)

CEDN - Common Extensible Design Notations

David Wile (Teknowledge) and David Garlan (CMU) were co-chairs of this committee whose scope broadly included most issues of specifications, specifically architecture styles and constraint specification, broad-based architectural interchange, architectural dynamism, and event specifications. For specifying architectural styles and especially, constraints, some community consensus was arrived at using the language Armani [Monroe], developed at CMU by Bob Monroe. Irvine's xADL group [van der Hoek and Dashofy] influenced the Acme development of xARCH [Schmerl], an XML-based architecture description formalism for tool interchange. David Wile's proposal for dynamic Acme [Wile '01c] was accepted for describing dynamism in Acme, but funding ran out before significant retooling of the Acme Tool Suite could be done to make use of the specifications. Event specification was never agreed upon completely, primarily because concerns for what should constitute a transaction were never wrung out, but David Wile (Teknowledge) proposed a unified approach to describing architecturally significant events [Wile '02] (see attachment). Again, this proposal would almost certainly have been adopted had the DASADA Phase II follow-on been funded.

Technology Transfer

The key technology developed under DASADA funding that was ready for transfer is the Probe bus technology. This was mature enough to be used on multiple platforms, as we illustrated in the Safe Email application. Unfortunately, most DASADA community members are not running Windows-based platforms, so even beta testing in the community was impossible, let alone finding real customers.

But the primary problem with transferring even the probe technology was the lack of an over-arching framework such as that proposed in the DASADA Phase II follow-on. The use of probes in isolation from this framework makes little sense. Had the architecture model-based infrastructure demonstrated in the Safe Email reformulation (described above) matured as projected in that follow-on proposal, the technology would be much more attractive to users requiring less technical savvy than the potential customers of the current technology would need.

References

[Allen] R. Allen. A Formal Approach to Software Architecture. Ph.D. Thesis. Carnegie Mellon University CMU Tech. Report CMU-CS-97-144, May, 1997.

[Balzer '01b] Balzer, R.: "Assuring the Safety of Opening Email Attachments," Proceedings of the DARPA DISCEX Conference, Anaheim, California, June 2001, pp.1257.

[Balzer and Goldman] Balzer, R. and Goldman, N.: "Mediating Connectors: A Non-ByPassable Process Wrapping Technology," Proceedings of the DARPA DISCEX Conference, Hilton Head, South Carolina, 2000, pp.361-368.

[Balzer '01a] Balzer, R.: "The DASADA Probe Infrastructure," Technical Report, Teknowledge Corp. (available from authors), 2003.

[Carzaniga] Carzaniga A., Rosenblum D. S., and Wolf A. L.: Achieving scalability and expressiveness in an Internet-scale event notification service. ACM Transactions on Computer Systems (TOCS) 19(3), 2001, 332-383.

[Crane et al] Crane, S., Dulay, N., Fossa, H., Kramer, J., Magee, J., Sloman, M., and Twidle, K.: "Configuration Management for Distributed Systems," Proceedings of the IFIP/IEEE

International Symposium on Integrated Network Management (ISINM), Santa Barbara, California, 1995.

[Dashofy, van der Hoek, and Taylor] E. Dashofy, A. van der Hoek, and R. Taylor A Highly-Extensible, XML-Based Architecture Description Language. In Proceedings of The Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 2001

[Egyed and Wile] Egyed, A. and Wile, D.: "Statechart Simulator for Modeling Architectural Dynamics," Proceedings of the 2nd Working International Conference on Software Architecture (WICSA), August 2001, pp.87-96.

[Feather] M.S. Feather. Language Support for the Specification and Development of Composite Systems, ACM Transactions on Programming Languages and Systems 9(2), April, 1987, 198-234.

[Garlan, Monroe, and Wile] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In Proceedings of CASCON'97 (November, 1997). (See also: http://www.cs.cmu.edu/~acme)ACM SIG PROCEEDINGS template. http://www.acm.org/sigs/pubs/proceed/template.html.

[Garlan and Schmerl] Garlan, D. and Schmerl, B.: "Model-based Adaptation for Self-Healing Systems," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), Charleston, South Carolina, November 2002, pp.27-32.

[Garlan, Schmerl, and Chang] Garlan, D., Schmerl, B., and Chang, J.: "Using gauges for architecture-based monitoring and adaptation," Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

[Goldman and Balzer] N. Goldman and R. Balzer. The ISI Visual Design Editor Generator. IEEE Symposium on Visual Languages. Tokyo. Sep. 1999. 20-27.

[Luckham et al] D. Luckham, et al. Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering 21(4) April, 1995. Pp. 336-355. (See also: http://anna.stanford.edu/rapide/).

[Magee and Kramer] J. Magee and J. Kramer. Dynamic structure in software architectures. In Proceeding of the ACM SIGSOFT '96: Fourth Symposium on the Foundations of Software Engineering. San Francisco, CA Oct., 1996. Pp. 24-32.

[Minsky] N. Minsky. Independent on-line monitoring of evolving systems. ICSE 96, Berlin, Germany. Pp. 134-143.

[Monroe] Monroe, R.: "Capturing software architecture design expertise with Armani," Technical Report CMU-CS-98-163, Carnegie Mellon University, 1998.

[Moriconi] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. IEEE Transactions on Software Engineering 21(4) Apr. 1995. Pp. 356-372.

[Narayanaswamy] K. Narayanaswamy. http://www.darpa.mil/ito/Summaries97/D931_0.html

[P. Oreizy, Medvidovic, andTaylor] P. Oreizy, N. Medvidovic, R. Taylor. Architecture-Based Runtime Software Evolution. Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 19-25, 1998, 177-186.

[Schmerl] B. Schmerl. Acme. http://www.cs.cmu.edu/~acme

[Schmerl and Gross] B. Schmerl and P. Gross. DASADA Architecture Mutation Schema. http://www.cs.columbia.edu/~phil/dasada/DASADA-Arch-Mutation-Schema.html

[Shaw] M. Shaw, et al. Abstractions for software architecture and tools to support them. IEEE Transactions of Software Engineering 21(4) Apr., 1995. pp. 314-335.

[Valleto and Kaiser] Valleto, G. and Kaiser, G.: "A Case Study in Software Adaptation," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), Charleston, South Carolina, November 2002, pp.73-78.

[van der Westhuizen and van der Hoek] C. Van der Westhuizen and A. van der Hoek. Understanding and Propagating Architectural Changes. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture. 2002.

[Wile '00] D. Wile. The EDCS Architecture Legacy. September 11, 2000. Santa Fe, NM. (slides available from author).

[Wile '01a] Wile, D. S.: "Residual Requirements and Architectural Residue," Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE 2001), Toronto, Canada, August 2001, pp.194-201.

[Wile '01b] Wile D. S.: Modeling Architecture Description Languages Using AML. Automated Software Engineering Journal 8(1), 2001, 63-88.

[Wile '01c] D. Wile. Using Dynamic Acme. In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture. Brisbane. Dec. 2001.

[Wile '02] Wile, D.: "Towards a Synthesis of Dynamic Architecture Event Languages," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), Charleston, South Carolina, November 2002, pp.79-84.

[Wile '03] Wile, D. "Calculating Requirements: an Approach Based on Architecture Style." In proceedings of the 2003 Automated Software Engineering Conference. Montreal. Oct.

[Wile and Egyed] Wile, D. and Egyed, A. An Externalized Infrastructure for Self-Healing Systems." In proceedings of the Working IEEE/IFIP Conference on Software Architecture. Oslo. June, 2004. (To appear).