AFRL-IF-RS-TR-2004-324
Final Technical Report
November 2004

DEFINITION, DEPLOYMENT AND USE OF GAUGES TO MANAGE RECONFIGURABLE COMPONENT-BASED SYSTEM

University of Colorado

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K511

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

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-2004-324 has been reviewed and is approved for publication

APPROVED: /s/
ELIZABETH S. KEAN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 2004	**FINAL** **Jun 00 – Jun 04**

4. TITLE AND SUBTITLE
DEFINITION, DEPLOYMENT AND USE OF GAUGES TO MANAGE RECONFIGURABLE COMPONENT-BASED SYSTEM

5. FUNDING NUMBERS
G - F30602-00-2-0608
PE - 62302E
PR - DASA
TA - 00
WU - 10

6. AUTHOR(S)

Dennis Heimbigner
Alexander Wolf

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

Computer Science Department
430 UCB
University of Colorado
Boulder CO 80309-0430

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/IFSA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-324

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Elizabeth S. Kean/IFSA/(315) 330-2601

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

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

The University of Colorado DASADA project provides an application architecture and associated automated infrastructure (FIRM) that can support the dynamic reconfiguration of a software system. Its intended use is to insert sensor probes and actuators into a target system and to support the reconfiguration of the target to improve its operation based on data reported by the probes. The FIRM infrastructure includes: a general communication substrate based on content-based routing (Siena); a standardized interface for manipulating components (LIRA); a content-directed control system (Q/A); a hypertext system for data accumulation and correlation (InfiniTe); a runtime configuration management system (TWICS); and finally, a planning system to support complex reconfigurations (Planit).

14. SUBJECT TERMS	15. NUMBER OF PAGES
Dynamic Reconfiguration, Runtime Configuration Management, Content Directed Control System	43
	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

List of Tables

Preface

As stated in the original BAA, the goal of the DASADA Program was to develop new technology to the development of complex distributed systems. It intended to provide assurance that the assembly of components comprising the distributed system could operate as expected. The issue standing in the way was complexity. Complexity resulted from the following:

- The partial understanding of the relationship between physical structure and behavior of the system. The system was composed from large number of types of components and connections and the overall system had only weak central control.
- The behavior of the distributed system was actually the composition of multiple, essential, interdependent behaviors (e.g. to support functional, safety, reliability, security requirements).
- The system exhibited dynamic behavior in both component behavior and topology; survivability and changing operating conditions required adaptation and self-correction.

DASADA was based on the assumption that it was not feasible to completely model such systems. Since successful system implementation is based on having models that are accurate with respect to critical properties, the consequence was that it appeared to be difficult or even impossible to understand, predict, control, automatically compose, or adapt such systems.

The goal, then, of DASADA was to circumvent the lack of models by focusing on the runtime behavior of such systems. The idea was to define important properties of the system and to check and enforce those properties at runtime. This approach was not intended to replace modeling but rather to complement it and to provide support when the model was (as it almost always is) incomplete. In fact, DASADA maintained a strong focus on software architecture to provide an important class of specialized models that could be used to support runtime adaptation and survivability. As stated in the BAA:

> "… As systems become more complex, it is evident that they must be able to change themselves by swapping or modifying components, changing component interaction protocols, or changing topology. They must do this dynamically, while the system is operating. The DASADA vision is that we can either have or create (through design or design recovery) a description of a system's architecture, a specification of critical properties (which may or may not be supported by the present system), and requirements for change. We should be able to refine or expand the architecture, guaranteeing those properties that can be guaranteed through design rules and adding components that are required to guarantee properties at run time. We should be able to (1) automatically (re)compose the system while it is operating (if necessary) to provide the needed capabilities and (2) continue monitoring operations to assure that the system continues to function properly. We should be able to use these techniques to modify distributed and heterogeneous systems, and should have the capabilities to assure properties of "off the shelf" or "open source" components with respect to the requirements of a specific system.

> We must be able to demonstrate these capabilities in a component-based world, with multiple standard communication (e.g., DCOM, CORBA, DCE) or "structuring" (e.g., eXtended Markup Language (XML), Resource Description Framework (RDF), Document Object Model (DOM)) infrastructures."

The approach taken generally by DASADA was to support the insertion of gauges into running systems and to develop the infrastructure and methodologies needed to support the use of gauges. The various projects within DASADA then addressed themselves to various parts of this problem.

It is fair to say that the DASADA program was quite prescient in recognizing the need for general tools for monitoring and controlling distributed systems. As direct outgrowths of DASADA (via participants) two recurring workshops, RAMSS (Remote Analysis and Monitoring of Software Systems) and WOSS (Workshop on Self-healing Systems) have continued to explore the research area first staked out by the DASADA program.

1. Introduction

The original objectives of the University of Colorado DASADA project were three-fold:

1. Provide the necessary infrastructure to effectively deploy, activate, and reconfigure the components of a distributed application;
2. Provide the infrastructure to insert probes and gauges into deployed applications;
3. Provide the infrastructure to capture, fuse, and disseminate the outputs of probes and gauges. As we shall see, this infrastructure is based on the Siena content-based routing (CBR) system. By the end of the DASADA program, Siena had become the def-facto data communication mechanism for transmitting probe and gauge data streams.

A note on terminology is in order. Soon after the initiation of the DASADA program, the concept of gauge was broken into two concepts: probes and gauges. A probe is a very lightweight sensor component that is inserted into an application to produce output measuring some property of the application. A gauge is an aggregator of the output of probes and other gauges to produce additional data with additional semantics. Thus, the general scheme is to insert probes into the application and then to construct a network of gauges to utilize a combination of streams of data from the probes and from other gauges. This is illustrated in Figure 1.

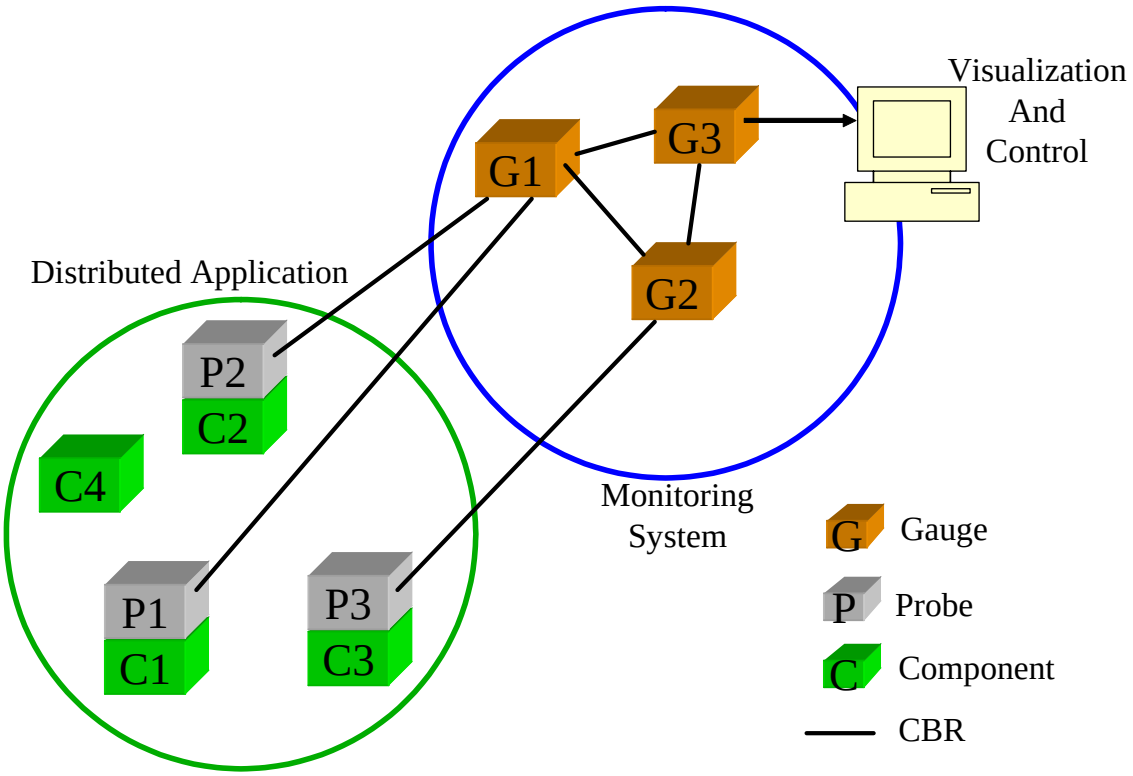

Figure 1. Example Probe and Gauge Network

Reconfiguration is a center element in the DASADA approach to the gauging of distributed systems. It is the basis for inserting (and removing) probes and gauges into systems to be monitored. Further, configuration information about an application also provides a useful and effective class of data for which interesting gauges can be constructed. This data can be used to reduce configuration and deployment failures when fielding new or upgraded software systems.

Our objective is to provide a comprehensive infrastructure supporting a complete life cycle including initial deployment of distributed applications, a sequence of reconfigurations of those applications, and finally the orderly shutdown of the application. A secondary objective of our work is to demonstrate the use of probes and gauges by integrating gauge data with heterogeneous collections of design artifacts and system models to improve consistency between design artifact intentions and fielded system reality.

In close association with our ability to insert probes and gauges, the University of Colorado is also providing the necessary communication mechanism to support the collection, fusion, and dissemination of the probe and gauge outputs. An important aspect of this mechanism is the need for loose coupling of the information providers and consumers so that, for example, new probes and gauges can be instantiated and their output automatically directed to any interested consumer. Further, this mechanism is extended to manage the probes and gauges to, for instance, control the granularity or periodicity of outputs. Finally, this communication mechanism must be able to support large-scale dissemination where there may large numbers of producers, consumers, and messages, and this support must provide maximum flexibility and near-optimal performance.

2. Method

The University of Colorado initially proposed to develop four systems to meet our objectives for this project: FIRM to address reconfiguration, Siena to address information dissemination, InfiniTe to address integration of gauge outputs with development-time information (e.g., design information), and TWIC to provide run-time configuration management. During the course of the project, we recognized some weaknesses in our initial proposal and we addressed these deficiencies by extending our work to include three more projects: Query/Advertise to support control of target systems and probes, LIRA to provide a standard interface between components and the monitoring system, and Planit to support complex reconfigurations by using AI planning techniques. The following section provides an overview of all of these systems.

3. Technical Results

3.1 FIRM

FIRM is our name for our deployment and reconfiguration infrastructure. It is unique in providing a detailed, automatically maintained, integrated set of models of a system's component, dependency, and run-time interconnection structure. These models are coupled to the actual system by means of an automated, agent-based configuration and deployment infrastructure that is itself distributed to support the deployment and reconfiguration of distributed applications. FIRM is an extension of our existing Software Dock system developed under the EDCS program and is incorporating the architecture models from the University of California, Irvine xADL project.

The FIRM framework provides a common infrastructure for deploying and reconfiguring component-based systems, for deploying probes and gauges, and for capturing, fusing, and disseminating the outputs of probes and gauges. Figure 2 depicts the architecture of the infrastructure as instantiated in some hypothetical situation. We use this figure to introduce the major elements of our technical approach.

At the top left of the figure is the data capture component, InfiniTe (Section 3.5). This component is responsible for capturing the output of gauges and providing a repository for inter-relating the captured data with other information such as design or architecture information.

At the top right is the configuration management (CM) component (Section 3.7) and the (re-) configuration service (Section 3.4). The CM component maintains information about the current state of the target application, including, for example, what probes are installed and what components are currently instantiated. The reconfiguration service is used by CM to construct a reconfiguration plan to move the target system from its current configuration to a new configuration.

At the bottom of the figure are two field sites, which may or may not be residing at remote network locations within a computing enterprise. At each site is an activated system, consisting of some arrangement of interconnected, executing components. The activated system is an executing instance of a deployed system, which was previously configured and installed at the field site and waited to be activated. Before being installed, the system’s components, along with its associated agents, were imported from one or more of the depots and cached at the site. An activated system can itself be a subsystem of a larger, system of systems. Notice in the figure, for example, that two of the components in one of the systems are interacting with a third component that is part of the other system. Thus, the distinction between “system” and “subsystem” is not useful in this context, so we simply use the term “system” to refer to both at all levels of abstraction.

Reconfiguration involves a series of coordinated agent activities, including component configurations, installations, activations, and deactivations, possibly taking place at multiple field sites. The plan produced by the reconfiguration service defines the specific actions and their

(partial) order. The reconfiguration actions are executed at a given site by the field reconfiguration controller (FRC). The FRC is responsible for carrying out the reconfiguration activities.

For activated systems, the agents must interact with executing components to carry out their activities. To do so in a generic fashion, they use the LIRA Standard Reconfiguration Interface (SRI) (Section 3.6), which we define as part of the FIRM framework and which all components, including probes and gauges, are expected to implement. Note that providing this interface for non-compliant COTS components may, in general, require the use of a wrapper technology. In any case, the SRI provides an abstraction of the primitive reconfiguration steps provided by facilities commonly found in configurable distributed object system frameworks.

The communication model underlying FIRM is that of loosely-coupled, event interaction supported by what is referred to as a Content-Based Routing (CBR) service. Our service is based

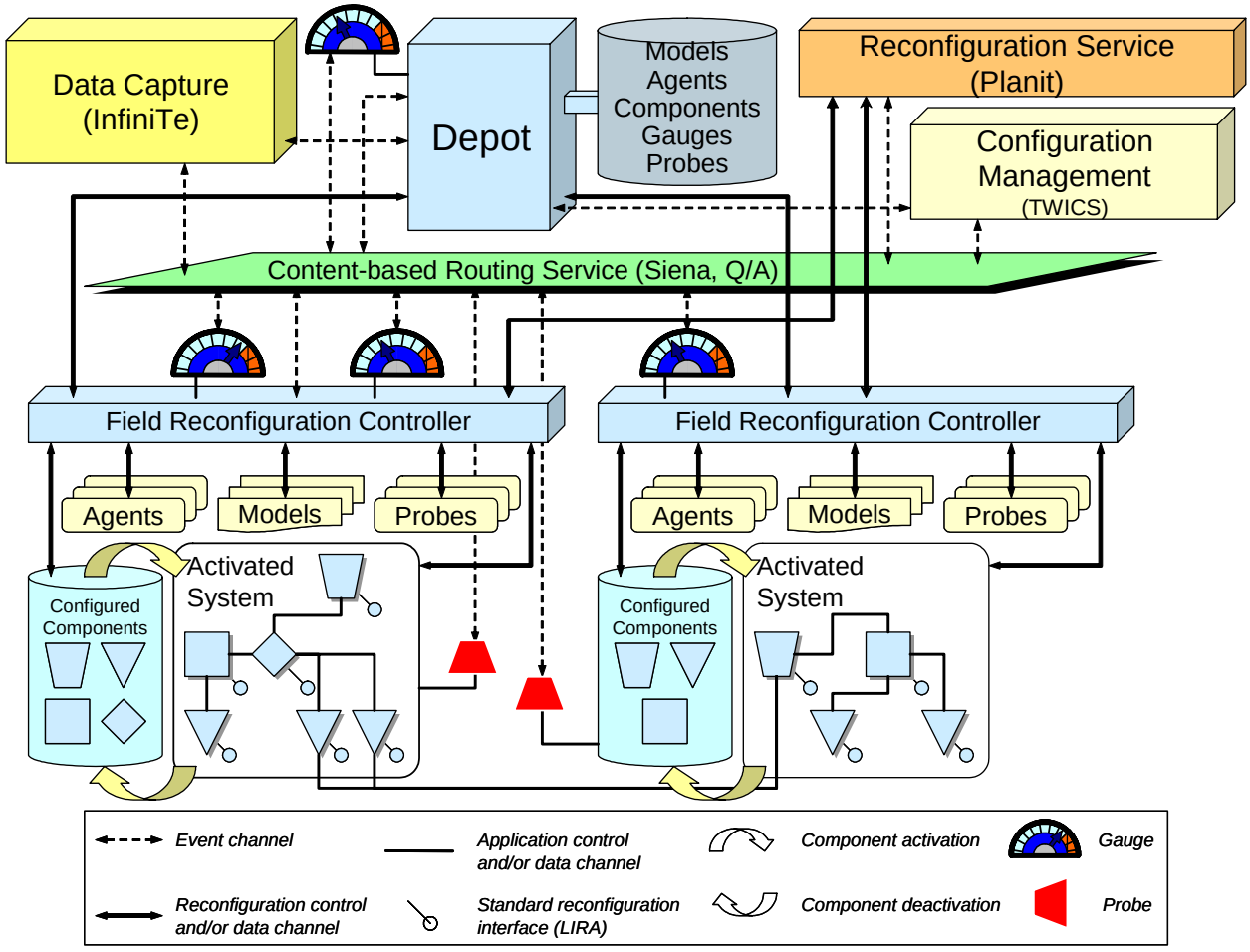

Figure 2. FIRM Architecture

on our Siena system (Section 3.2). CBR uses an extension of the well-known publish/subscribe

protocol to capture, fuse, and disseminate information throughout an enterprise. Notice in particular that the outputs of gauges are feed through the event notification service. This provides a uniform mechanism to integrate the handling of information from a disparate and dynamically changing population of gauges. In addition, control communication from the data gathering and configuration services are carried out through the CBR. We use a novel mechanism called query/advertise (Q/A) (Section 3.3)

At the top center of the figure is a depot. The depots can be thought of as repositories for the results of the development process, containing COTS components ready and waiting for configuration and deployment. Also found in the depots are LIRA agents (Section 3.6) that are used to automate the various activities related to managing a component-based system, including but not limited to information gathering, configuration, and deployment. Gauges and probes form another category of depot element, but in fact they are embodied as particular agents or as particular components to be deployed, and are thus treated uniformly within FIRM. The depot is thus a general repository used by many other components. It does itself provide little FIRM functionality, but is an enabler for other functions.

Each of the components in the figure is discussed in more detail in the following sections.

3.2 Siena

The Siena content-based routing middleware system developed at the University of Colorado was a significant contribution to DASADA. It became the de-facto standard for event distribution for almost all DASADA projects. By providing a common distribution mechanism, Siena significantly contributed to the interoperability of DASADA projects.

Content-based routing – also known as publish/subscribe – is a service in which the flow of messages, from senders to receivers, is driven by the content of the messages, rather than by explicit addresses assigned by senders and attached to the messages. Using a content-based communication service, receivers declare their interests by means of selection predicates, while senders simply publish messages. The service consists of delivering to any and all receivers each message that matches the selection predicates declared by those receivers.

Content based routing provides a powerful mechanism for integrating probes and gauges. It allows probes to generate events without regard to who will receive them and it allow gauges to specify the events of interest without too much regard for which probes are generating those events.

Siena messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. In standard Siena, the set of supported types is bool (true or false), long (64-bit integer), double (128-bit floating point), and byte-string, which also subsumes the more traditional string type. An example message could be represented as shown in the left column of Figure 3.

Message	Filter
{(artist, “Flatt and Scruggs”) (title, “Speedball”) (bitrate, 256) (MP3, true)}	{(artist,*, “Scruggs”) (bitrate, >=, 256)}

Figure 3. Example Message and Filter

A client constructs a filter (a pattern) that specifies the kinds of messages it wishes to receive based on the message content. A filter is a set of triples of the form (attribute, operator, value). A filter matches a message if the value associated with each attribute in the message satisfies all corresponding filter triples that have the same attribute name. That is, for a given filter F and a given message M, the following holds.

$$\forall \text{ triples } (x,op,a) \in F \qquad (1)$$
$$(\forall \text{ pairs } (y,b) \in M$$
$$(x = y \Rightarrow \text{Apply}(op,a,b)=\text{true}))$$
$$\text{where Apply}(op,a,b) = (a\ op\ b)$$

The set of filter triples may be considered to be logically “and”ed together. A logical “or” can be achieved by specifying multiple separate filters. The right side of Figure 3 shows an example filter that would match the message on the left side. Table 1 shows the complete set of pre-defined operators available in standard Siena. Since they are used for matching, they all produce a boolean result.

It is important to note that the attribute names used in messages and filters have no inherent semantic meaning. As with all such attribute-based systems, there must be some external agreement about their meaning, and all parties must adhere to that agreement.

Siena adopts a two-tier peer architecture where arbitrary Siena routers connect to form a specific topology. In the simplest case, a client connects to a router and provides one or more filters. The router then forwards the filter to all of its peers. Each peer records where the filter came from, and forwards it to its peers. Later, when some other client connects to a router and generates a message, the local copy of the filter can be applied at that router to determine the next router to whom the message should be forwarded. If a message is generated for which no filter matches at the local router, then it will not be forwarded at all and so will generate no inter-router traffic. This kind of content-based routing is analogous to IP routing in the Internet, but instead of specific IP addresses, the content of messages determines the destination (or destinations) for the message.

As part of the DASADA effort, we designed and implemented a number of improvements with respect to the baseline Siena available at the beginning of the project.

Table 1. Siena Filter Operators

Operator}	Argument Type
Equals (=)	bool, long, double, byte-string
Not-Equals (≠)	bool, long, double, byte-string
Less-Than (<)	long
Greater-Than (<)	long
Less-Equals (<=)	long
Greater-Equals (>=)	long
Prefix (>*)	byte-string
Suffix (*<)	byte-string
Contains (*)	byte-string
Any (any)	N.A.

3.2.1 Siena-XML

A number of the projects in the DASADA program told us that they would like to have an XML-based interface to Siena. The idea was to provide a mechanism by which the events could be generated from XML documents. This turned out to be non-trivial because of the desire on our part to maintain the existing event format. Switching to XML would have caused significant performance degradation because of the need to parse XML documents at every router and to provide some form of matching for XML documents. Both operations were deemed very costly.

Siena-XML represents a compromise in which an XML document is presented to Siena-XML and it converts it to a Siena event for standard dissemination. This had two advantages. First, Siena itself – the internal routing algorithms – did not require any modification; hence XML and non-XML events could co-exist on the same Siena network. Second, the conversion to and from XML was performed at the client. This meant that only the clients using Siena-XML would pay a price for its convenience.

Specifically, the Siena-XML project had three goals.

1. Extend Siena to accept publisher data in XML. XML is more expressive than the name-value pairs that make up attributes in Siena notifications. It is also possible that XML data will be easier for a publisher to generate, if the publisher's data is already represented internally as XML.
2. Allow a Siena subscriber to define a mapping from XML data to notifications and their attributes. The subscriber will still only receive data in the form of notifications, whether the publisher sends data as XML or notifications. The expressiveness of XML should be preserved as much as possible by allowing the subscriber to define a mapping between selected pieces of the XML data and both notifications and attributes. It is important to note that different subscribers to the same published XML data can have different mappings.

3. At runtime, use subscriber XML mappings to generate notifications from XML data. This is merely the runtime glue that maps the first two requirements together. The system should take published XML data and use the subscriber mappings to generate Siena events.

Siena-XML consists of two major components, the Siena-XML Designer and the Siena-XML Runtime. The user can define a mapping from XML documents to Siena events using the Siena-XML Designer. This definition is then stored in a Mapping File (in XML format). The Siena-XML Runtime actually performs the mapping from XML data to Siena events.

The runtime translation from XML to a standard event form basically operated as follows.

1. Specific fields in the XML document are extracted using XSLT or XPath (an XML query language) as specified during the Designer phase.
2. These extracted values are associated with specific attributes in the event using an additional specification.
3. The extracted (attribute,value) pairs are inserted into the event.
4. The actual XML document is inserted into the event with a special attribute name.

Because of step 4, it was possible for the receiver to obtain the complete XML document, as well as the extracted attribute values.

Siena-XML was tested and refined through interactions with the projects that originally requested it. The version described here came into general use and has proven to be quite satisfactory for those projects.

3.2.2 Siena Fast-Forwarding

As in traditional address-based networks, the delivery function for Siena is performed incrementally by passing messages between intermediate nodes in the network. We say that messages flow from upstream nodes to downstream nodes until delivered. The delivery function consists of two interrelated sub-functions: routing and forwarding. Routing establishes flow paths through the network by compiling and positioning local forwarding tables at each node. A forwarding table contains the information necessary for a node to decide to which neighbor node or nodes a given message should be sent; the processing of a message at a node is the forwarding sub-function. Taken together, the forwarding performed at the nodes causes messages to be routed through the network.

The general architecture of a content-based network's router is depicted in Figure 4. Following traditional networking terminology, we say that a router communicates with each neighbor node through an interface. The forwarding function processes an incoming message, consults the forwarding table, and determines the set of interfaces on which to output the message.

The key feature of the new routing algorithm is that it pre-compiles the routing tests into a network (Figure 5). Routing then consists of passing the message to be routed through the network and following edges based on the contents of the message and the tests at each node of

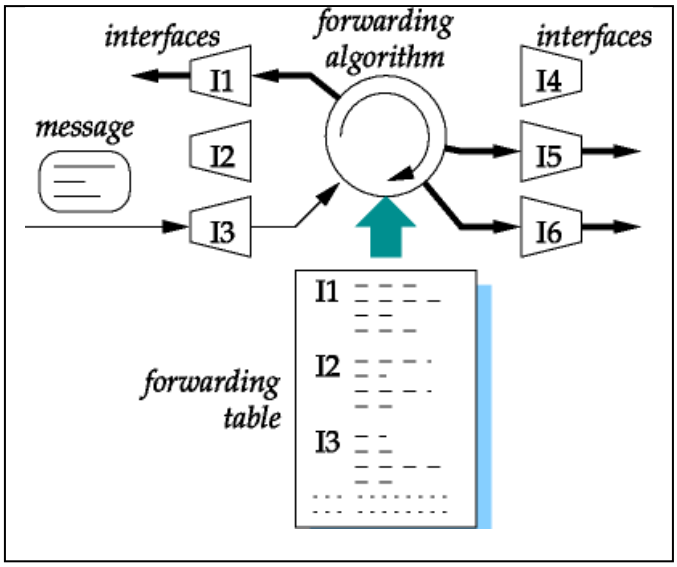

Figure 4. Fast-Forwarding Architecture

the network. When a terminal node is reached, either the message will not be forwarded or the node indicates where to forward the message.

Our evaluation shows that the algorithm has good absolute performance under heavy loads and in a variety of network configurations, including the extreme case of a single, centralized router. It also shows that the algorithm scales sub-linearly in the number of filters, with almost no degradation of throughput in the context of a network of routers with a fixed number of neighbor nodes. For example, on a 950MHZ processor, the algorithm is able to forward a 10-attribute message in 3 milliseconds in a situation where there are 20 predicates (i.e., neighbors) consisting of 250000 filters formed from 5 million individual constraints over an alphabet of 1000 attributes. In this experiment, the message went to 18 of the 20 neighbors, but we observed in other experiments that the performance generally improves (i.e., the forwarding time goes down) as the percentage of matching neighbors goes down. In terms of space, the forwarding table in this experiment occupies only 48 bytes per constraint, even though we have not yet turned our attention to optimizing that aspect of the algorithm.

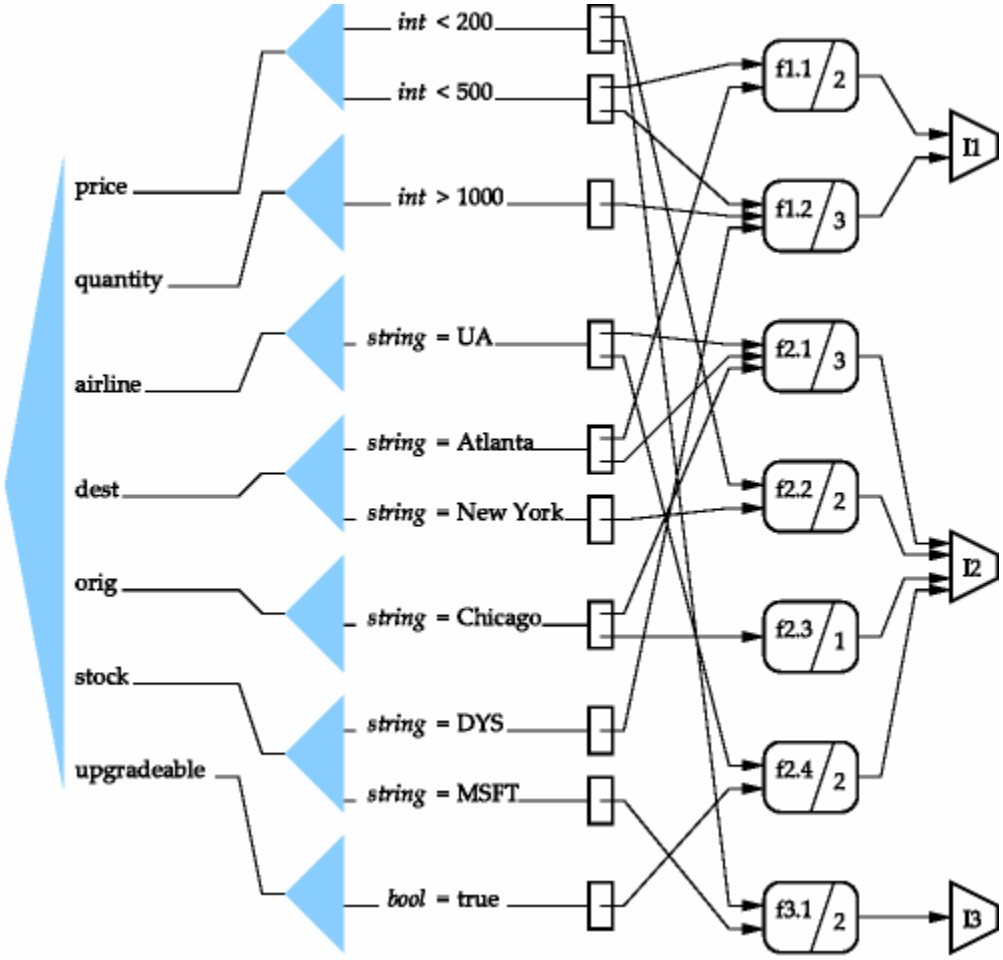

Figure 5. Example Fast-Forwarding Network

3.2.3 Siena Version 1.5

During this project, a major new release of Siena was published: release version 1.5.0 of the Java-based API and server. This release included some bug fixes and a number of important improvements:

- Simplified client-master protocol. Eliminated WHO/INF exchange between clients and masters. This change simplifies the initial client-master connection reducing the number of messages and removing the need to have globally unique identifiers.
- Added notification input/output sequencer. Sequencers can be used to deliver notifications in the order they were published.
- Fixed and improved monitoring facility, now fully compatible with the Siena Monitor.
- Various small corrections and improvements to Makefile and tests/Makefile.

3.3 Query/Advertise

Effective implementation of large-scale application control requires the ability to communicate from the monitoring components of a FIRM control system to the various application nodes in the network. This communication is in two forms: query, and control. As part of this project, we developed a novel extension to the content-based routing system to provide both query and control in a relatively efficient form.

The approach involves exporting configuration-relevant properties from each component (or combination of components on a single host machine). Later, queries can be broadcast using CBR to locate components with specified properties, such as which components have a particular probe active. The same mechanism can be used to broadcast property-based reconfiguration and control notifications. These notifications are targeted by property values so that they reach exactly the set of relevant components. This kind of intentional addressing avoids the need to manage detailed lists of all components and supports effective control of dynamically changing sets of components.

3.3.1 Query

In the query model, an advertiser is a client of the query/advertise system who "advertises" the availability of some kind of information using a special kind of query that describes the data available at that client. Other clients issue queries that are distributed to each advertiser whose advertisement is deemed to "match" the query. It is the job of the query/advertise system to ensure that queries are efficiently directed only to those data sources that may have information matching the query.

Upon receiving the query, the advertiser applies it to its local data and responds with the resulting data. The response may be returned through the publish/subscribe network but it may be returned using some other mechanism such as a point-to-point TCP connection. The net effect is that the original query client receives, from multiple sources, data that matches its query. That client can then collate the responses to produce an aggregated result. This whole process involves a sequence of advertise-query-respond combinations, but we will refer to this simply as query/advertise (Q/A)

For comparison purposes, recall that in CBR (publish/subscribe) systems (Section 3.2), clients publish notification (or event) messages with highly structured content. Other, subscribing, clients make available a filter (a kind of pattern) specifying the subscription: the content of notifications to be received at that client. It is the job of the publish/subscribe system to ensure that notifications are efficiently delivered to the clients with matching subscriptions.

Publish/subscribe and query/advertise are in a sense duals of each other. A subscription represents a way for a site to indicate that specified notifications should be routed to the subscriber. An advertisement represents a way for a site to indicate that specified queries should be routed to the advertiser. Similarly, a publisher sends out notifications that should be routed to matching subscribers. A queryer sends out queries that should be routed to matching advertisers.

This duality is important because query/advertise is mapped onto publish/subscribe by mapping advertisements to subscriptions and queries to notifications.

Query/advertise has the notion of a response that is inherent in any system for querying data sources, but which has no dual in publish/subscribe. Responses may in fact be provided without using the publish/subscribe system at all. Therefore mapping the response mechanism to publish/subscribe requires some special handling. The simplest solution, and the one used for FIRM, is to encode the sender as a special field in the message and then let the responder send out a reply that indicates that sender.

3.3.2 Control

The query/advertise system can be extended so that it can also serve to control large numbers of components, even when detailed knowledge about all the affected components is unknown or difficult to maintain.

The idea is that we extend our query messages to include a payload of control information. This payload might include probe code modules or parameters. The query is used to specify the kinds of target clients to whom the payload is directed. Thus, one could send out a control message whose query matches Linux-based systems and whose code probe is specific to Linux. A separate message could be disseminated for Windows-based hosts.

3.4 Planit

The Planit research was started relatively late in project. It was in response to our recognition that there was need for a significantly better approach to the dynamic reconfiguration of both target software as well as the monitoring systems that inserted probes and managed gauges.

This new project has as its goal the application of AI planning to the problem of configuration and reconfiguration of complex distributed, component-based systems. We have developed a novel technique for carrying out the deployment and reconfiguration planning processes that leverages recent advances in the field of temporal planning. This is embodied in a tool called

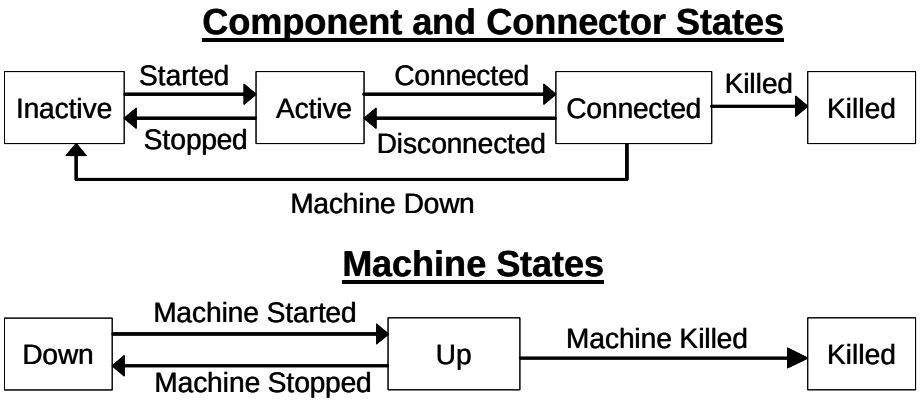

Figure 6. Planit Runtime Architecture Model

Planit, which manages the deployment and reconfiguration of a software system utilizing a temporal planner. Given a model of the structure of a software system, the network upon which the system should be hosted, and a goal configuration, Planit will use the temporal planner to devise possible deployments of the system. Given information about changes in the state of the system, network and a revised goal, Planit will use the temporal planner to devise possible reconfigurations of the system.

Planit requires three artifacts to find a plan:

1. The domain encodes the semantics of the system. The domain provides a model of the classes of components and a set of constraints between the components. These constraints include inter-component dependencies. Figure 6 shows the gross structure of our model. The model is derived from the UC, Davis xADL architecture model. Note that we extend that model to include the hardware – the specific machines – upon which the software executes.
2. The initial state describes the state of the system at the present moment.
3. The goal state describes the desired state of the system.

The domain is fixed and can not be changed at runtime. However, the initial state and goal states are variable and are determined by the state of the system after a failure. In order to standardize the planning terminology and to exchange and evaluate results, the AI planning community has developed a standardized language for defining the domain, initial state and goal state. This language is called PDDL (Planning Domain and Definition Language), and we have adopted this language for our system.

3.4.1 Dynamics of the Planning Process

The non-static elements of planning are the initial state and the goal state of the system. They are dependent on the individual scenario, so they must be constructed at runtime. Since a change of one component may have a ripple effect in the system, a preprocessing step is required to compute the set of other components affected by the change. After this preprocessing the initial and goal states are constructed and given to the planner to find a plan (a sequence of actions) to get from the initial state to the goal state.

The basic procedure for applying planning to a failure is as follows:

1. Check if a component has failed. This can be done either by polling of machines and components, or by some form of event notification.
2. For the failed component (including machines), calculate its ripple effect using a dependency model.
3. Construct the “system model” for each of the failed components. The system model is a set of predicates and functions describing the component’s current state. The system model becomes part of the initial state for the planner.
4. Choose the goal configuration.

5. Invoke the planner with the domain, initial and goal states as input and obtain a plan in return.
6. Execute the plan for system recovery

3.4.2 Example

In this example, we have six non-client components deployed on five different machines. These components form a layered hierarchy based on inter-component dependencies. These six components are a web server, two servlet engines, two application servers and a database. Moreover, there are clients connected to the web server. There are internal clients that support customer service and maintenance, and there are external clients supporting customers.

The distributed system is depicted in Figure 7. All the rectangles represent machines where instances of these components are working. All machines but one has a single instance of the component working. The instances of the servlet engine and the application server are not redundant. They have distinct sets of servlets and EJBs respectively. The application server also has a built-in servlet engine, but in order to provide better performance, separate servlet engines are installed. However, the built-in servlet engines in the application servers can be used if required.

The types of failures possible in this system are failures of the component or failure of the machine. In order to demonstrate our approach we assume that our sensors and detectors inform us about failure of machine 2. Further, we make the assumption that the failure of machine 2 is catastrophic and it can not be recovered in a reasonable amount of time.

The failure of machine 2 has a number of effects on the overall health of the distributed system.

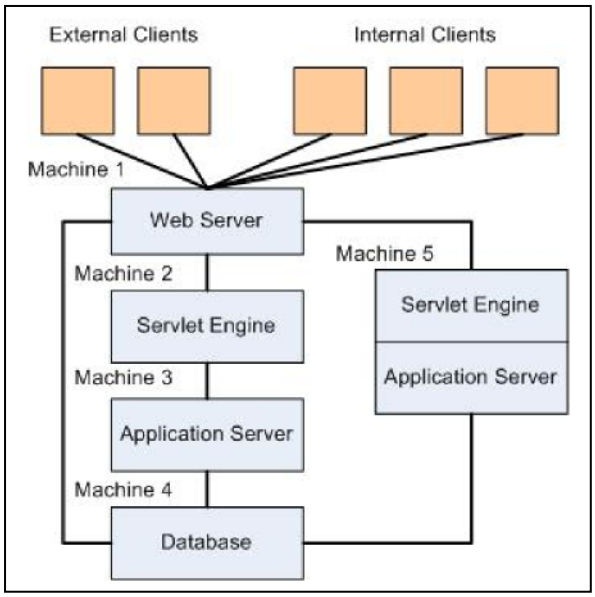

Figure 7. Example

First of all the servlet engine on machine 2 also fails because of the explicit physical dependency. Second, the servlet engine failure causes a ripple effect preventing the application server on machine 3 from delivering its functionality. Note that the application server is still running but it is unable to respond to any client requests because of its dependency on the servlet engine. Third, there is a set of clients that also lose their connection because of the failure. Other artifacts in the system continue to work as normal; the failure is not total.

The task of failure recovery is to bring the total functionality of the distributed system back online. The recovery process must not impact existing performance and functionality in the other parts of the system.

In order to recover the system there are many configurations that can be adopted. However, these configurations have different cost, time and resource implications. The goal of planning is to find the best alternate amongst these configurations. There are two general classes of recovery configurations that restore the system from failure taking into account the complete loss of machine 2.

1. Deploy the servlets that were in the failed engine into another servlet engine and connect that engine with the web server. There are two choices for the alternate servlet engine: either the servlet engine on machine 5 or the built-in engines associated with the application servers on machines 3 or 5.
2. Install a new instance of the servlet engine onto another machine (i.e. 1, 3, or 4) and then start it and connect it to the application server on machine 3.

The above failure recovery scenarios have different cost, time and resource implications. It may also be the case that we need a hybrid of the above approaches. This is especially true when there are some servlets that are critical and that needs to be back up online as soon as possible, while other servlets can wait for a certain period of time. Moreover, these target configurations require various steps to be carried out in a specific order before the system reaches a workable state. In order to make our discussion simple but concrete we will only consider the last set of scenarios that assume that the lost servlet engine will be restarted on some other machine and that all the lost servlets will be deployed into that engine.

A subset of the domain for our example system is depicted in pseudo-PDDL in Figure 8. The domain description has three parts: predicates, functions and actions.

Actions generally correspond to the interfaces in present day Architecture Definition Languages (ADLs), xArch in our case. Actions are the set of operations that can be executed on the system. Each action has a duration. The duration is defined as a function in the functions list. The numerical value of the duration is specified in the initial state. The preconditions of the actions are a set of logical expressions over the predicates and functions. In general all preconditions that involve a number are defined using functions and all preconditions that are boolean are defined using predicates. Predicates also serve the purpose of specifying the state of each component in the system (i.e., the equivalent of ground assertions in, say, Prolog). All predicates by default are false unless specifically asserted in the initial condition.

```
Objects(applicationserver machine webserver servletengine)

Predicates
ServletEngineInstalled(servletEngine, machinename)
ServletEngineStarted(servletEngine)
ServletEngineWorking(servletEngine)
machineFailed(machinename)
ApplicationServerWorking(applicationServer)
WebServerWorking(webserver)
...
Functions
MachineRAM(machinename)
MachineStartTime(machinename)
ServletEngineInstallTime(servletEngine)
ServletEngineConnectTimeWithWS(servletEngine)
...
;; Actions
Start-Machine(machinename)
Duration (= (MachineStartTime(machinename))
Preconditions
        (not (machineFailed machinename))
effects
        machineStarted(machinename)
Install-Servlet-Engine(servletEngine machinename)
Duration (= (ServletEngineInstallTime(servletEngine))
PreCondition
        (> (MachineRAM(machinename) 512)
        (= (MachinePlatform(machinename) Unix)
        (= (MachineJDK (machinename) 1.4.2)
effects
        ServletEngineInstalled(servletEngine)

Start-Servlet-Engine (servletEngine)
Duration (= (ServletEngineStartTime(servletEngine))
PreConditions
        ServletEngineInstalled(servletEngine)
Effects
        ServletEngineStarted(servletEngine)

Connect-ServletEngine-AS(servletEngine, applicationserver)...)
Connect-ServletEngine-WS(servletEngine, webServer)...)
ServletEngineWorking...
```

Figure 8. Example Domain Specification

An action can only take place if the desired preconditions are fulfilled. If the preconditions are not fulfilled additional actions must be taken to fulfill those preconditions. Each action has a set of effects. The effects are represented in terms of functions and predicates also. The effects may change a boolean value of a predicate or change a numerical value in a function. The effects in general represent the new state of the system.

Planit adds a dependency model as a new element of the domain and a new element in planning. The dependency model is used to check the extent of damage after a failure. It is represented as a graph that determines the dependencies amongst the components. The dependencies also take into account the states of the component in which the dependency becomes effective. For example the applications server and servlet engine are dependent on each other because they need to be connected for the dependency to be effective. However, they can not be connected unless both of them are in a started state. If the connection is lost then both of them will change state and the dependency will lose its effectiveness. The dependency will still hold, however, and in order for the system to work normally the dependency must be restored. The dependency model also takes into account hardware dependencies such as the machine or network availability. This means that the hardware resources in the system are also part of the dependency model.

The initial state (really the current state) is the second input to the planner (after the domain model) It contains the list of all the components present in the system, their names, state and properties For instance the function MachineRAM(machineName) in the domain will specify the RAM available on a certain machine in the initial state using an expression such as the following:

=(MachineRAM(machine1) 512))

The goal state is the third input to the planner. It specifies the desired configuration after the failure. There are two ways in which a goal state can be represented: Implicit State and Explicit State. In an implicit state the predicates that need to be true are specified in the goal state and it is up to the planner to find the best configuration possible and the best set of steps to reach that configuration. In an explicit state, however, an explicit configuration is given and the planner's job is to select the best set of steps to reach that defined explicit configuration. Given its three inputs, our planner constructs a plan for reconfiguring the system from its current state to the new goal state.

3.5 InfiniTe

Software engineers confront many challenges during software development. One challenge is managing the relationships that exist between software artifacts. We refer to this task as *information integration*, since establishing a relationship between documents typically implies that an engineer must integrate information from each of the documents to perform a development task. This integration is hindered because software artifacts are created with a multitude of tools and are stored in a variety of data formats, and none (or few) of these tools were designed to interoperate or make it easy to share information. We have developed (and continue to develop) an information integration environment, *InfiniTe*, to aid software developers in performing complex information management tasks. In particular, we focus on supporting those tasks which involve creating, finding, maintaining, and evolving the relationships (both implicit and explicit) between software artifacts.

InfiniTe is an important element of FIRM because it provides the repository for storing information about probes and gauges and about the structure of the target systems being monitored by FIRM. Although not accomplished in this project, TWICS would have been connected to InfiniTe to server as configuration repository. In addition, probe and gauge outputs provide information about runtime relationships between component objects, and this information can be stored in InfiniTe.

InfiniTe is designed to support the construction of new relationships between previously unrelated information sources. It uses an Open Hypermedia system to store and manage those new relationships, and it used a mediation mechanism to make new relationships visible in existing documents where viewers for those documents whose viewers provide extensible mechanisms for linking to fragments of the document. For DASADA, we used gauges as sources of information that could be related to existing design documents about the application from which the gauge data was being extracted. This provides the potential to relate, for example, real-time performance information with the requirements originally established for the application.

An early version of InfiniTe existed before the DASADA project. This early version used an XML-based repository for storing information. As part of DASADA, we migrated the InfiniTe environment to a structure server provided by the Themis structural computing environment. A structure server is essentially an object-relational database that has been augmented with APIs to allow the definition and manipulation of application-specific data structures to occur at a higher level of abstraction through the use of a powerful structure template mechanism.

We additionally developed a requirements traceability framework, TraceM, which is layered above InfiniTe and our open hypermedia system. TraceM provides services that are specifically tailored to support engineers in performing requirements traceability. Work during this period has focused on developing integrators and translators to discover and maintain requirements traceability information over requirements documents in Microsoft Word, design documents in

ArgoUML, source code, and Web-based mailing lists. In addition, we developed two integrators that are used by TraceM to monitor the changes made to sets of InfiniTe relationships over time.

3.6 LIRA

An important issue in the Willow framework concerns the mechanisms for telling applications to reconfigure. Reconfiguration may be divided into two kinds: internal and external. Internal reconfiguration relies on the programmer to build into a component the facilities for reconfiguring the component. This form of reconfiguration is sometimes called “programmed” or “self-healing” reconfiguration. External reconfiguration, by contrast, relies on some entity external to the component to determine when and how the component is reconfigured. Internal and external reconfiguration are closely linked because, in the end, almost all external reconfiguration relies on internal reconfiguration capabilities; the external entity determines when and which internal reconfiguration is to be invoked.

LIRA represents our attempt to construct a standard, decentralized interface between internal reconfiguration mechanisms and external reconfiguration commands as required by the Willow architecture. The inspiration for our approach comes directly from the field of network management and it’s Internet-Standard Network Management Framework, which for historical reasons is referred to as the Simple Network Management Protocol (SNMP). Our hypothesis was that this framework could serve, with appropriate extension and adaptation where necessary, as a useful model for lightweight reconfiguration of component-based, distributed software systems.

The essence of the approach we take in LIRA is to define a particular method for applying the basic facilities of the Internet-Standard Network Management Framework to complex component-based software systems. To summarize:

- We distinguish two kinds of agent. A reconfiguration agent is associated with a component, and is responsible for reconfiguring the component in response to operations on variables defined by its Management Information Base (MIB). A host agent is associated with a computer in the network, and is responsible for installing and activating components on that computer, again, in response to operations on variables defined by its MIB.
- A manager can itself be a reconfiguration agent. What this means is that a manager can have a MIB and thereby be expected to respond to other, higher-level managers. Such a manager agent would reinterpret the reconfiguration (and status) requests it receives into management requests it should send to the agents of the components it is managing. In this way a scalable management hierarchy can be established, finally reaching ground on the base reconfiguration agents associated with (monolithic) components.
- We define a basic set of “standard” MIB definitions for each kind of agent. These definitions are generically appropriate for managing software components, but are expected to be augmented on an agent-by-agent basis so that individual agents can be specialized to their particular unique tasks.

Table 2. LIRA Protocol

Request	Response
SET(variable name, variable value)	ACK(message text)
GET(variable name)	REPLY(variable name, variable value)
CALL(function name, parameters list)	RETURN(return value)
	NOTIFY(variable name, variable value, agent name)

It is important to note that LIRA does not itself provide the agents, although in our prototype implementation we have created convenient base classes from which implementations can be derived.

3.6.1 Reconfiguration Agent

A base reconfiguration agent directly controls and manages a component. LIRA does not constrain how the agent is associated with its component, only that the agent is able to act upon the component. For example, the agent might be part of the same thread of execution, execute in a separate thread, or execute in a completely separate process. In fact, the agent might reside on a completely different device, although this would probably be the case only for complex agents associated with components running on capacity-limited devices.

A reconfiguration agent that is not a base reconfiguration agent is a manager. It interacts with other base and non-base reconfiguration agents using the standard management protocol. For purposes of simplifying the discussion below, we abuse the term “component” to refer also to the subassembly of agents with which a non-base reconfiguration agent (i.e., a manager agent) interacts. Thus, from the perspective of a higher-level manager, it appears as though a lower-level manager is the same as any other reconfiguration agent. A reconfiguration agent is essentially responsible for managing the lifecycle of its component, and exports at least the management functions shown in Table 2.

Shutdown is a function that also serves to terminate the agent. LIRA provides the notion of a “function” as a convenient shorthand for a combination of more primitive concepts already present in SNMP.

3.6.2 Host Agent

A host agent runs on a computer where components and reconfiguration agents are to be installed and activated, and is responsible for carrying out those activities in response to requests from a manager. (How a host agent is itself installed and activated, is obviously a bootstrapping process.) As part of activating a component and its associated agent, the host agent provides an available network port, called the agent address, to the reconfiguration agent over which that agent can receive requests from a manager.

3.6.3 Management Protocol

The management protocol (Table 2) follows the SNMP paradigm. Each message in the protocol is either a request or a response. Requests are sent by managers to agents, and responses are sent

back to managers from agents. The last message, NOTIFY, is sent from agents to managers in the absence of any request. This message is used to communicate an alert from an agent back to a manager. For instance, an agent might notice that a performance threshold has been crossed, and uses the alert to initiate some remedial action on the part of the manager.

3.7 TWICS

Early on we recognized that configuration management of components was going to be an important element of our infrastructure. Automatic reconfiguration demands it in order to have a standardized way to obtain components to augment, replace, upgrade, and repair the components our distributed applications.

André van der Hoek's group at the University of California at Irvine was the designated researcher to address this problem. His approach was called TWICS, which stands for "Two-Way Integrated Configuration management and Software deployment." TWICS is built as a layer on top of Subversion, an existing open source configuration management system that supports the traditional code development paradigm. TWICS adds a component-oriented interface and wraps all the Subversion commands to operate on components as a whole rather than individual source files. Moreover, TWICS adds deployment functionality in the form of an interface through which the release of components and their dependencies can be controlled, and through which remote components can be downloaded and placed in the local repository. Finally, TWICS respects the different levels of autonomy, privacy, and trust that may exist among different organizations. To do so, it allows individual organizations to establish and enforce policies that govern such issues as to whether a component can be obtained in binary or source form and whether a component may be redistributed by other organizations.

Figure 9 illustrates the architecture of TWICS. To support this integration, TWICS builds upon an existing configuration management system (Subversion) and overlays it with functionality for

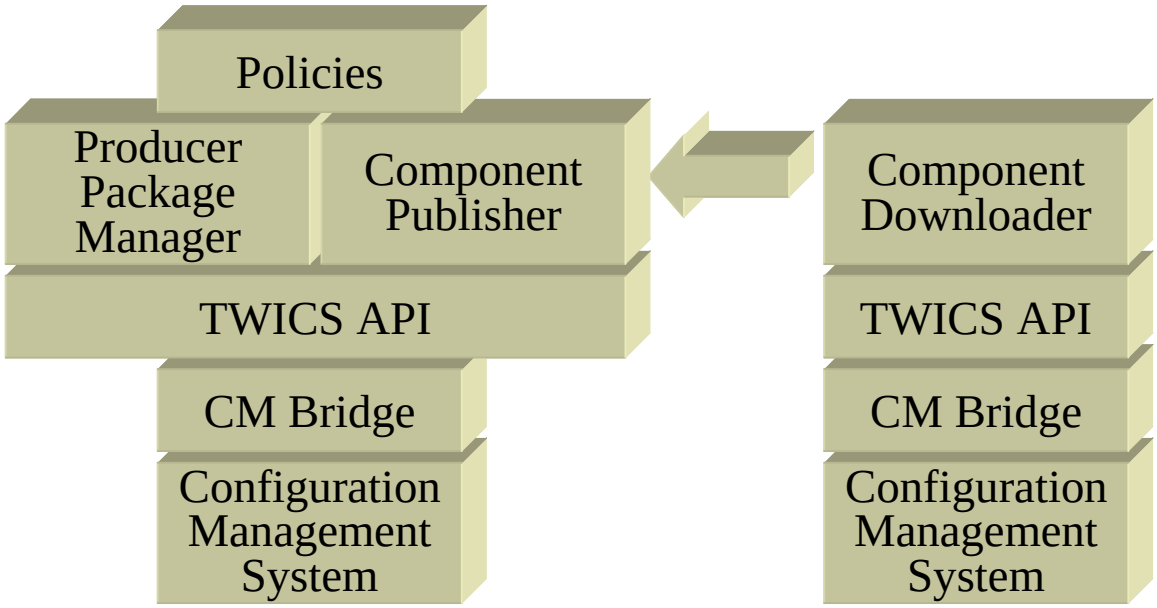

Figure 9. TWICS Architecture

explicitly managing and deploying components. In particular, TWICS organizes the configuration management repository in a component-based fashion and adds three extra modules for managing the resulting artifacts:

- A *producer package manager*, which supports a developer in publishing a component,
- A *component publisher*, which supports other organizations in downloading a component, and
- A *component downloader*, which downloads a component and places it under local configuration management control.

All these modules take into account the possibility of dependencies. Additionally, they collaborate in establishing and enforcing different trust policies as they exist among different organizations.

3.7.1 Policies

The policies component in Figure 9 plays a key role in this architecture. It contains the declarative data describing the trust policy of a particular organization. Upon a download request, the component publisher consults the policies component to ensure that only trusted parties are able to download the components.

3.7.2 Repository

TWICS configures the repository structure of its underlying configuration management system in such a way as to enable a transparent development and deployment of both internal and external components. TWICS partitions the available versioning space into two parts: a separate module called “TWICS”, in which it stores all the data pertaining to its own operation, and the remaining versioning space, in which actual development takes place.

To preserve integrity, users should not directly manipulate the TWICS part of the repository, but use the TWICS user interface instead. The remaining versioning space, however, can be manipulated at will, provided that users adhere to a naming scheme in which each Subversion module represents a component.

The TWICS part of the repository serves as a controlled store for packaged components. These components may have been developed locally, in which case they are stored for download by remote organizations. They may also have been developed remotely, in which case the controlled store serves as a staging area from which components are unpacked and brought into the development side of the repository. Components are stored as archives, and may be available as a source archive, a binary archive, or both. The metadata of a component is stored alongside the component itself, containing both data describing the component (e.g., name, version, authoring organization, dependencies) and data concerning the trust policy to be applied (e.g., organizations permitted to download the component, redistribution permissions).

Using the native versioning mechanisms of Subversion, different versions of a component can be stored in the TWICS repository. This allows different versions of a locally developed component

to be available for download, and furthermore supports an organization in obtaining and separating multiple versions of an externally developed component.

Separating the TWICS repository from the actual development area helps to separate the remote evolution of an external component from any local changes that may have been made. When a new version of such an external component is downloaded, placing it in the TWICS repository does not disturb any local development. At a later time, suitable to the local organization, can the remote changes be merged and integrated into the local effort.

Note that, under this scheme, different instances of TWICS in effect act as peers to each other. Each can simultaneously serve as an instance that makes components available to other instances and as an instance that downloads components from other instances.

3.7.3 Producer package manager

Once a component has been developed using normal configuration management procedures (e.g., using Subversion), it must be packaged and made available for release. To do so, a developer uses the TWICS *producer package manager* component. This component first requests some metadata from the developer describing the component. In particular, it requests the name and version of the component to be released, the server to which it should be released (defaulted to the current configuration management repository), and any dependencies that the component may have on other components. After this information has been filled out, TWICS automatically creates a source and a binary distribution for the component and places them on the TWICS side of the repository. Currently, TWICS assumes Java components and creates a package by automatically including source files in the source package and class files in the binary package. Future versions of TWICS will extend this capability significantly.

3.7.4 Component Publisher

Once the trust policy has been specified, the component is available to other organizations for download. The download process itself is governed by the *component publisher* component, whose sole responsibility is to make sure the consumer is allowed the request they are making. If not, the component cannot be downloaded. If the request can be granted, the component, along with its metadata, is shipped to the consumer.

3.7.5 Component Downloader

The *component downloader* component eases the process of obtaining remotely developed components. Based on the input received from the user, the component downloader sends a deployment request to the producer of the requested component. If the user is indeed allowed to download the component, an archive will be received from the producer and placed on the TWICS side the configuration management repository. After that, the component downloader checks the metadata of the component and automatically fetches any required dependencies. This process is performed recursively, until all the dependencies are obtained and placed in the local

environment. Note that if a component that is a dependency is already at the consumer site, it will not be downloaded again.

After the components have been downloaded, TWICS supports a user in automatically unpacking the archives and placing the components on the development side of the repository. From there on, a user can start using the components. Note that if a user locally modified a component for which they are now downloading a new version, TWICS currently requires them to manually integrate the two (perhaps with assistance from the standard merge routines in Subversion). While TWICS makes sure to not overwrite any changes, automated support for detecting and resolving these situations is certainly desired (see Section 3.5).

A first working version of the TWICS configurable architecture support system has been completed and is now available as a prototype. TWICS is built on top of Subversion, an open-source configuration management system, and overlays it with functionality for component-based configuration management in a decentralized setting. In particular, TWICS allows different organizations to collaborate by establishing trust policies that garner which organizations may access which components in which form. An automated download manager "installs" components from another organization into the local configuration management repository, and accounts for any dependencies that may have to be downloaded as well.

4. Software

All of the above systems have associated prototype implementations. These can be obtained from the following points of contact.

André van der Hoek (andre@ics.uci.edu)	TWICS
Antonio Carzaniga (carzanig@cs.colorado.edu)	Siena, LIRA
Dennis Heimbigner (dennis@cs.colorado.edu)	Planit, Query/Advertise, FIRM
Ken Anderson (kena@cs.colorado.edu)	InfiniTe

5. Program Management

The project team was a consortium of two universities: the University of Colorado at Boulder and the University of California at Irvine. The investigators at each University were in the Computer Science Departments of their respective universities:

- University of Colorado, Boulder: Alexander Wolf, Dennis Heimbigner, Kenneth Anderson
- University of California, Irvine: André van der Hoek

6. Technical Transfer

As part of our effort in DASADA, we explored a number of approaches to transferring the technology to various internal and external consumers.

6.1.1 Internal Consumers

Siena has become the de-facto standard communication substrate within the DASADA program, and it is used by almost every project. This has helped to provide an important level of interoperability between DASADA projects.

In the course of DASADA, we also engaged in a number of shorter term collaborative exercises to integrate our technologies with other DASADA projects. These included the following.

- GeoWorlds: USC/ISI made their GeoWorlds system available. We used GeoWorlds as the basis for a comprehensive demonstration that showed the use of a number of our technologies. This demonstration illustrates the generation of Siena event notices based on data and actions provided by GeoWorlds. These data were in XML and demonstrated our XML front end for Siena, SXML. The events were received by the InfiniTe system and used to show the integration of GeoWorlds data with other information. In addition, we developed a configuration description of GeoWorlds. This allowed us to apply our Configuration Fitness Gauge to GeoWorlds.
- AWACS: UCI, Lockheed Martin, Boeing, USC, and Colorado explored possible ways to demonstrate the gauging, deployment, and dynamic reconfiguration of the new AWACS architecture. This subsequently led to a joint DASADA Phase II proposal.
- Troop Deployment: We coordinated with USC to apply our FIRM technology to their troop deployment demonstration to show dynamic reconfiguration of a network of fixed and mobile devices in support of a troop deployment exercise.

6.1.2 External Consumers

We participated in two DASADA Phase II proposals:

1. A proposal for an AWACS-based demonstration for DASADA Phase II was completed. This involved UCI, Lockheed Martin, Boeing, USC, and Colorado.
2. A proposal with TACOM and the University of Massachusetts.

Unfortunately, neither was selected, but they did provide a vehicle for exploring tech transfer issues.

A number of applications of Siena were explored.

- Dr. Carzaniga explored (and continues to explore) the possibility of obtaining a patent on the Fast-Forwarding technology. This is in conjunction with the University of Colorado

Intellectual Property Rights organization. This is in response to some commercial interest in the technology.

- Our Siena fast forwarding algorithm was used as a basis for a business plan by a team of MBA students from the Leeds School of Business of the University of Colorado at Boulder. This transfer activity has continued and we have involved the University of Colorado technical transfer office to see if any of this work can be commercialized. The same plan was also selected to participate in a national competition. Local technology venture capital investors also expressed interest in the technology and in the business plan.
- We coordinated with the University of California, Santa Barbara, on the HI-DRA project supported by the Army Research Laboratory. HI-DRA is a project to develop a network-level intrusion detection system. We provided Siena to perform the event notification services for disseminating intrusion alarms. This work has been an important driver for improving the efficiency of Siena because packet-level sniffing inherently requires high throughput.

We proposed the use of the Planit reconfiguration domain for use in the 2004 International Planning Competition (http://ipc.icaps-conference.org/), which is part of the 2004 International Conference on Automated Planning and Scheduling. This was important because helped to focus the AI planning community on this problem and should result in better planners that we can incorporate into our Planit system. Unfortunately, it did not reach the finals of selection. We plan to try again in the future.

We also actively worked to introduce our technologies into a number of projects outside of DASADA. In particular, we successfully demonstrated the use of our reconfiguration technology and our event notification technology in the Air Force Joint Battlespace Infosphere (JBI) program. In September 2001, we traveled to Rome to demonstrate a reconfigurable instance of a JBI. JBI assumes a publish/subscribe paradigm, so Siena was used in studying its scalability. We also examined the use of reconfiguration as a means for making the JBI more survivable against non-maskable faults, threats, and intrusions. We also participated in the AFRL Science Review Board review held at Rome in October, 2001.

7. Acknowledgements

The material used in preparing this final report is derived from the original proposal as well as from the various publications (Section 8) produced by this project. The author of this report wishes to thank the authors of those publications for allowing the use of that material.

8. Publications

1. Anderson, K., S. Sherba, and W. Lepthien, "Towards Large-Scale Information Integration," Proc. of the 2002 Int'l Conf. on Software Engineering (ICSE-2002), Orlando, Florida, May 19-25, 2002, pp. 524-535.
2. Anderson, K., S. Sherba, and W. Lepthien, "Structure and Behavior Awareness in Themis," Proc. of the 2003 ACM Conf. on Hypertext, Nottingham, UK, August 26-30, 2003, pp. 138-147.
3. Anderson, K., S. Sherba, and W. Lepthien, "The Themis Structural Computing Environment: Structural Templates and Transformations," Journal of Network and Computer Applications: Special Issue on Structural Computing (to appear).
4. Anderson, K. and S. Sherba, "Using Open Hypermedia to Support Information Integration," Proc. of the Seventh Int'l Workshop on Open Hypermedia Systems.
5. Anderson, K., A. Carzaniga, D. Heimbigner and A. Wolf. “Event-based Document Sensing for Insider Threats,” Department of Computer Science Technical Report CU-CS-968-04, University of Colorado, February 2004.
6. Arshad, N., D. Heimbigner, and A. Wolf, "Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems," Proc. of the fifteenth IEEE Int'l Conf. on Tools with Artificial Intelligence, November 3-5, 2003, Sacramento, CA.
7. Arshad, N., D. Heimbigner, and A. Wolf, "A Planning Based Approach to Failure Recovery in Distributed Systems," Proc. of the ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04), Newport Beach, CA, 31 Oct. - 1 Nov. 2004 (to appear).
8. Caporuscio, M., A. Carzaniga, and A. Wolf, "An Experience in Evaluating Publish/Subscribe Services in a Wireless Network," Proc. of the third Int'l Workshop on Software and Performance, in conjunction with Int'l Symposium on Software Testing and Analysis (ISSTA), Rome, Italy. July 2002.
9. Caporuscio, M., A. Carzaniga, and A. Wolf, "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications," IEEE Transactions on Software Engineering 29(12):1059-1071, December 2003.
10. Carzaniga, A., D. Rosenblum, and A. Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service", Proc. of the nineteenth ACM Symposium on Principles of Distributed Computing, July, 2000, Portland OR.
11. Carzaniga, A., D.S. Rosenblum, and A. L. Wolf. “Design and Evaluation of a Wide-Area Event Notification Service”, ACM Transactions on Computer Systems 19(3):332-383 (August 2001).
12. Carzaniga, A., and A. L. Wolf, “Content-Based Networking: A New Communication Infrastructure”, NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Lecture Notes in Computer Science 2538, pp. 59-68, Springer-Verlag, Berlin, 2002.

13. Carzaniga, A. and A. Wolf, "Fast Forwarding for Content-Based Networking," Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado, November 2001. Revised September 2002.

14. Carzaniga A. and Alexander L. Wolf, "Content-Based Networking: A New Communication Infrastructure," NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Lecture Notes in Computer Science 2538, Springer-Verlag, Berlin, 2002, pp. 59-68.

15. Carzaniga, A. and A. Wolf, "Forwarding in a Content-Based Network," Proc. of SIGCOMM 2003, Karlsruhe, FRG, August 2003, pp. 163-174.

16. Carzaniga, A., M. Rutherford, and A.L. Wolf, "A Routing Scheme for Content-Based Networking," Proc. of IEEE INFOCOM 2004, Hong Kong, China. March 2004.

17. Castaldi, M., A. Carzaniga, P. Inverardi, and A. L. Wolf, “A Lightweight Infrastructure for Reconfiguring Applications”, Technical Report CU-CS-943-02, Department of Computer Science, University of Colorado, December 2002.

18. Dashofy, E., A. van der Hoek, and R. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," Proc. of the Working IEEE/IFIP Conf. on Software Architecture, Amsterdam, Netherlands. September 2001, pp. 103-112.

19. Dashofy, E. and A. van der Hoek, "Representing Product Family Architectures in an Extensible Architecture Description Language," Proc. of the Int'l Workshop on Product Family Engineering, Bilbao, Spain, October, 2001.

20. Dashofy, E., A. van der Hoek, and R. Taylor, "An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages," Proc. of the twenty-fourth Int'l Conf. on Software Engineering, May 2002, pp. 266-276.

21. Dashofy, E., A. van der Hoek, and R. Taylor, Towards Architecture-Based Self-Healing Systems," Proc. of the first ACM SIGSOFT Workshop on Self-Healing Systems, November 2002.

22. Dincel, E., N. Medvidovic, and A. van der Hoek, "Measuring Product Line Architectures," Proc. of the Int'l Workshop on Product Family Engineering, Bilbao, Spain, October, 2001.

23. Estublier, J., D. Leblang, G. Clemm, R. Conradi, A. van der Hoek, W. Tichy, D. Wiborg-Weber, "Impact of the Research Community for the Field of Software Configuration Management," Proc. of the twenty-fourth Int'l Conf. on Software Engineering, May 2002, pp. 643-644.

24. Heimbigner, D., “Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality”, 2001 ACM Symposium on Applied Computing (SAC 2001), pp. 176-181, Las Vegas, NV, 11-14 March 2001.

25. Heimbigner, D., "An Efficient Implementation of Query/Advertise," Technical Report CU-CS-948-03, Department of Computer Science, University of Colorado, March 2003.

26. Heimbigner, D., "A Tamper-Detecting Implementation of Lisp," Proc. of the 2003 Security and Management Conf., June 23-26, 2003, Las Vegas, NV.

27. Heimbigner, D., "Common Issues for Remote Analysis and Adaptive Security," Proc. of the second Int'l Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04), Edinburgh, Scotland, May 24, 2004.

28. Heimbigner, D., "DMTF - CIM to OWL: A Case Study in Ontology Conversion," Proc. of the Ontology in Action Workshop in conjunction with the 2004 Conf. on Software Engineering and Knowledge Engineering (SEKE'04), Banff, Alberta Canada, June 21, 2004.

29. Heimbigner, D., "Managing Access Rights for Terminated Employees," Proc. of the 2004 Int'l Conf. on Security and Management, Las Vegas, NV, June 2004.

30. Heimbigner, D., "Expressive and Efficient Peer-to-Peer Queries," Proc. of the 38th Hawaii Int'l Conf. on System Sciences (HICSS-38), Hawaii, HA, Jan. 3-6, 2005 (to appear).

31. Knight, J., D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu, “The Willow Survivability Architecture”, Proceedings of the Fourth Information Survivability Workshop, Vancouver, B.C, March 2002.

32. Konig-Ries, B., K. Makki, S. Makki, C. Perkins, N. Pissinou, P. Reiher, P. Scheuermann, J. Veijalainen, A. Wolf, and O. Wolfson, "Research Direction for Developing an Infrastructure for Mobile and Wireless Systems," NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Lecture Notes in Computer Science 2538, Springer-Verlag, Berlin, 2002, pp. 1-37.

33. Lüer, C., D. Rosenblum, and A. van der Hoek, "The Evolution of Software Evolvability," Proc. of the Int'l Workshop on the Principles of Software Evolution, Vienna, Austria. September 2001, pp. 131-134.

34. Oh Navarro, E. and A. van der Hoek, "Towards Game-Based Simulation as a Method of Teaching Software Engineering," Proc. of the Thirty-second ASEE/IEEE Frontiers in Education Conf., November 2002.

35. Reese, W., D. Heimbigner, and A. Wolf, "Pulling it all together with WIT: A tool for integrating Web-based Information", Proc of the Int'l Workshop on Information Integration on the Web (WIIW 2001), April 9-11, 2001, Rio de Janeiro, Brazil.

36. Rutherford, M., K. Anderson, A. Carzaniga, D. Heimbigner, and A. Wolf, "Reconfiguration in the Enterprise JavaBean Component Model," Proc. of the IFIP/ACM Working Conf. on Component Deployment, Berlin, FRG, 20-21 June 2002.

37. Sarma, A., and A. van der Hoek, "Palantír: Increasing Awareness in Distributed Software Development," Proc. of the 2002 ICSE Workshop on Global Software Development, May 2002, pp. 28-32.Temple, B., J. Brant, and M. Hauswirth, “SienaXML Project Summary”, 2003.

38. van der Lingen, R. and A. van der Hoek, “Dissecting Configuration Management Policies,” *Proceedings of the Eleventh International Workshop on Software Configuration Management*, Portland, Oregon, May 2003.

39. Sherba, S., K. Anderson, and M. Faisal, "A Framework for Mapping Traceability Relationships," Proc. of the second Int'l Workshop on Traceability in Emerging Forms of

Software Engineering (TEFSE'03), in conjunction with the 18th IEEE Int'l Conf. on Automated Software Engineering, Montreal, Quebec, October 7, 2003.

40. van der Hoek, A., M. Mikic-Rakic, R. Roshandel, and N. Medvidovic, "Taming Architectural Evolution," Proc. of the eighth European Software Engineering Conf., in conjuntion with the ninth Int'l Symposium on the Foundations of Software Engineering, Vienna, Austria. September 2001, pp. 1-10.
41. A. van der Hoek, "Integrating Configuration Management and Software Deployment," Proc. of the Working Conf. on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.
42. van der Hoek, A., A. Carzaniga, D. Heimbigner, and A. Wolf, "A Testbed for Configuration Management Policy Programming," IEEE Transactions on Software Engineering 28(1) (Jan 2002).
43. van der Hoek, A. and A. Wolf. "Software Release Management for Component-Based Software," Software-Practice & Experience (to appear).
44. van der Westhuizen C. and André van der Hoek, "Understanding and Propagating Architectural Change," Proc. of the Working IEEE/IFIP Conf. on Software Architecture 2002 (WICSA 3), Montreal, Canada, August 2002.
45. Wang, C., A. Carzaniga, D. Evans, and A. Wolf, "Security Issues and Requirements for Internet-scale Publish-Subscribe Systems," Proc. of the thirty-fifth Hawaii Int'l Conf. on System Sciences (HICSS-35), Hawaii, Hawaii, Jan. 2002.
46. Wolf, A., D. Heimbigner, A. Carzaniga, J. Knight, P. Devanbu, and M. Gertz, "Bend, Don't Break: Using Reconfiguration to Achieve Survivability", Proc. of the third Information Survivability Workshop (ISW-2000), Boston, MA, October 24-26, 2000, pp. 187-190.
47. Wolf, A., D. Heimbigner, K. Anderson, A. Carzaniga, and N. Ryan, "Achieving Survivability of Complex and Dynamic with the Willow Framework," Proc. of the Working Conf. on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

9. List of Acronyms

Acronym	Description
AFRL	Air Force Research Laboratory
AWACS	Air-born Early Warning and Control System
CBR	Content-based routing
CM	Configuration Management
FRC	Field Reconfiguration Controller
JBI	Joint Battlespace InfoSphere
MIB	Management Information Base
Q/A	Query/Advertise
SNMP	Simple Network Management Protocol
TWICS	Two-Way Integrated Configuration management and Software deployment
WWW	World Wide Web

10. Glossary

Term	Definition
Probe	A very lightweight sensor component that is inserted into an application to produce output measuring some property of the application.
Gauge	An aggregator of probe (and other gauge) data to produce additional data with additional semantics.
Content-based Routing	Content-based routing is the technology underlying publish-subscribe. It is a form of message routing in which the messages are distributed to various clients based on the contents of the messages and the client specified pattern describing the messages of interest. This contrasts with traditional IP routing, which is based on the use of specific addresses to route messages.