



CarnegieMellon
Software Engineering Institute

Discovering Architectures from Running Systems: Lessons Learned

CMU/SEI-2004-TR-016
ESC-TR-2004-016

Hong Yan
Jonathan Aldrich
David Garlan
Rick Kazman
Bradley Schmerl

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

December 2004

Software Architecture Technology Initiative

Unlimited distribution subject to the copyright.





**CarnegieMellon
Software Engineering Institute**

Pittsburgh, PA 15213-3890

Discovering Architectures from Running Systems: Lessons Learned

CMU/SEI-2004-TR-016
ESC-TR-2004-016

Hong Yan
Jonathan Aldrich
David Garlan
Rick Kazman
Bradley Schmerl

December 2004

Software Architecture Technology Initiative

Unlimited distribution subject to the copyright.

20051223 031

This report was prepared for the

SEI Joint Program Office
ESC/XPB
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Christos Scordas
Chief of Programs, XPB

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2005 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (<http://www.sei.cmu.edu/publications/pubweb.html>).

Table of Contents

Abstract.....	v
1 Introduction	1
2 Related Work	3
3 Technical Challenges	5
4 DiscoTest State Engine Design	7
4.1 State Machine Definition.....	7
4.1.1 Elements of a State Machine	7
4.1.2 Informal Operational Semantics.....	9
4.2 Pipe/Filter Example	11
5 Implementing DiscoTest	17
6 Adaptive Architectures for Mobile Systems (AAMS) Case Study	19
6.1 Design of the AAMS State Machine	20
6.2 The Discovered Architecture	21
7 EJB Case Study.....	23
7.1 Design of the EJB State Machine	23
7.2 The Discovered Architecture	24
8 Lessons Learned and Future Work.....	27
References.....	29

List of Figures

Figure 1: The DiscoTect Architecture	6
Figure 2: Elements of a State Map	8
Figure 3: State Machine for Discovering Filter Components.....	10
Figure 4: The State Machine Fragments for Discovering Pipe Connections	13
Figure 5: The Architecture Fragment Resulting from Running the System and Using the State Machine Shown in Figure 3.....	13
Figure 6: Relevant Output from the Event Filter	15
Figure 7: The Discovered Architectural Model of PrereqCheck	16
Figure 8: Documented Runtime View of AAMS	20
Figure 9: Discovered Architecture of AAMS	22
Figure 10: Documented Architectural View of Duke's Bank Application.....	24
Figure 11: Discovered Architecture of Duke's Bank.....	25

Abstract

One of the challenging problems for software developers is guaranteeing that a system as built is consistent with its architectural design. This report describes a technique that uses automatically generated runtime observations of an executing system to construct an architectural view of the system. In this technique, mappings are developed that exploit regularities in system implementation and architectural style. These mappings describe how low-level system events can be interpreted as more abstract architectural operations. In addition, this report describes the current implementation of a tool, called DiscoTect, that uses these mappings, and it shows how DiscoTect can highlight inconsistencies between implementations and architectures. Furthermore, two case studies are provided that illustrate how DiscoTect works and how it can be applied to real-world systems.

1 Introduction

For most complex systems, it is crucial to have a well-defined architecture. Such a definition provides a high-level view of a system in terms of its principal runtime components (e.g., clients, servers, databases), their interactions (e.g., remote procedure call [RPC], event multi-cast), and their properties (e.g., throughputs, reliabilities). As an abstract representation of a system, an architecture permits many forms of high-level inspection and analysis [Bass 03]. Consequently, over the past decade, considerable research and development has gone into the development of notations, tools, and methods to support architectural design.

Despite advances in developing an engineering basis for software architectures, a persisting difficult problem is determining whether a system as implemented has the architecture as designed. Without some form of consistency guarantees, the relationship between architectural insight and the actual system will be hypothetical at best, invalidating many of the benefits of architectural design.

Currently, two principal techniques have been used to determine or enforce relationships between a system's architecture and implementation. The first is to ensure consistency by construction. This can be done by embedding architectural constructs in an implementation language (e.g., as described by Aldrich and colleagues [Aldrich 02]) where program analysis tools can check for conformance. Or, it can be done through code generation, using tools to create an implementation from a more abstract architectural definition [Shaw 95, Taylor 96, Vestal 96]. Although it is effective when it can be applied, ensuring consistency by construction has limited applicability. In particular, it can usually be applied only in situations where engineers are required to use specific architecture-based development tools, languages, and implementation strategies. For systems that are composed of existing parts, or that require a style of architecture or implementation outside those supported by generation tools, this approach does not apply.

The second technique is to ensure conformance by extracting an architecture from a system's code, using static code analysis [Jackson 99, Kazman 99, Murphy 95]. When an implementation is sufficiently constrained so that modularization and coding patterns can be identified with architectural elements, this technique can work well. Unfortunately, however, the technique is limited by an inherent mismatch between static, code-based structures (such as classes and packages) and the runtime structures that are the essence of most architectural descriptions [Garlan 02]. In particular, the actual runtime structures may not even be known until the program runs: clients and servers may come and go dynamically; components (e.g., Dynamic Linked Libraries [DLLs]) not under direct control of the implementers may be dynamically loaded; and so forth.

A third, relatively unexplored, technique is to determine the architecture of a system by examining its behavior *at runtime*. The key idea is that a system can be monitored while it is running. Observations about its behavior can then be used to infer its dynamic architecture. This approach

- has the advantage that in principal it applies to *any* system that can be monitored
- gives an accurate image of what is actually going on in the real system
- can accommodate systems whose architecture changes dynamically
- imposes no *a priori* restrictions on system implementation or architectural style

There are a number of hard technical challenges in making this technique work. The most serious is finding mechanisms to bridge the abstraction gap: in general, low-level system observations do not map directly to architectural constructs. For example, the creation of an architectural connector might involve many low-level steps, and those actions might be interleaved with many other architecturally relevant actions. Moreover, there is likely no single architectural interpretation that will apply to all systems. Different systems will use different runtime patterns to achieve the same architectural effect, and, conversely, there are many possible architectural elements to which one might map the same low-level events. In this report, we describe a technique to solve the problem of dynamic architectural discovery for a large class of systems. The key idea is to provide a framework that allows the mapping of implementation styles to architecture styles. This mapping is defined as a set of conceptually concurrent state machines used at runtime to track the progress of the system and output architectural events when predefined runtime patterns are recognized. By parameterizing the framework by both architectural and implementation styles, we can exploit regularity in systems, while still providing flexibility in defining new abstraction mappings.

In this report, we introduce DiscoTect, a system for discovering the architectures of running systems. In Section 2, we discuss related work. Section 3 presents the technical challenges in producing an architecture discovery framework that can be used with multiple architectural styles and multiple systems. Section 4 presents our main technical contribution: the use of state machines to map between implementation-level events and architectural operations. We discuss implementing DiscoTect in Section 5, and we present results from two case studies to illustrate the utility of DiscoTect in Sections 6 and 7. In Section 8, we discuss the strengths and weaknesses of our approach. Finally, we present conclusions and future work.

2 Related Work

Our work is mostly related to other approaches for dynamic analysis of a system. A number of techniques and tools have been developed to extract information from a running system, including instrumenting the source code to produce trace information and manipulating runtime artifacts to get the information (e.g., as described by Balzer and Goldman [Balzer 99] and Wells and Pazandak [Wells 01]). Many technologies are available for monitoring systems, and we build on them. However, they do not, by themselves, solve the hard problem of mapping from code to more abstract models. In previous work, we developed an infrastructure doing certain kinds of abstraction [Garlan 03]. However, this approach was limited to observing properties of a system and reflecting them in a preconstructed architectural model. In the work discussed in this report, we show how to create that model.

Dias and Richardson [Dias 03] use an XML-based language to describe runtime events and to use patterns to map them into high-level events. Analyzing these events to determine architectural structure is not addressed. In addition, a simple static mapping from low-level system events to high-level ones has limited expressiveness. For example, it cannot handle the case where the event analyzer initially has an interest in one set of events, but then changes its interest after the initial events have occurred. Also, it doesn't provide a way of specifying event correlations or mapping a series of correlated low-level events to a single high-level event—a crucial capability needed when discovering the architecture of a system. Kaiser and colleagues use a collection of temporal state machines to perform pattern matching against runtime events [Kaiser 03]. Our approach is similar, but it makes architectural styles or patterns explicit.

A number of researchers have investigated the problem of presenting dynamic information to an observer. For example, some researchers present information about variables, threads, activations, object interactions, and so forth [Reiss 03, Walker 98, Walker 01, and Zeller 01]. Ernst and colleagues show how to dynamically detect program invariants by examining values computed during a program execution and by looking for patterns and relationships among them [Ernst 01]. This process is somewhat different from detecting architectural structure.

Madhav [Madhav 96] describes a system that allows Ada 95 programs to be monitored dynamically to check conformance to a Rapide architectural specification [Luckham 97]. His approach requires the source code to be annotated so that it can be transformed to produce events to construct the architecture. In contrast, our approach does not require access to the source code, and it does not rely on explicit architectural construction directives to be embedded in the code as does ArchJava [Aldrich 02].

A large body of research has investigated specification of the dynamic behavior of software architectures. Of the many approaches, some use explicit state machines (e.g., as described by Allen and Garlan [Allen 94] and Vieira and colleagues [Vieira 00]). These approaches, however, do not link architecture to an executing system.

3 Technical Challenges

Any approach that supports dynamic discovery of architectures must address three problems: (1) observing a system's runtime behavior, (2) interpreting that runtime behavior in terms of architecturally meaningful events, and (3) representing the resulting architecture. In this report, we are concerned primarily with the second problem of bridging the abstraction gap between system observations and architectural effects.

A number of issues make this a hard problem. First, mappings between low-level system observations and architectural events are not usually one to one. Many low-level events may be completely irrelevant. More importantly, a given abstract event, such as creating a new architectural connector, might involve many runtime events, such as object creation and lookup, library calls to runtime infrastructure, initialization of data structures, and so forth. Conversely, a single implementation event might represent a series of architectural events. For example, executing a procedure call between two objects might signal the creation of a new connector and its attachment to the runtime ports of the respective architectural components. This ambiguity implies the need for a technique that can keep track of intermediate information about mappings to an architectural model.

Second, architecturally relevant actions are typically interleaved in an implementation. For example, at a given moment, a system might be midway through creating several components and their connectors. Because architectural events are interleaved with each other, any attempt to recognize architectural events must be able to cope with concurrent intermediate states.

Third, there is no single gold standard for indicating what implementation patterns represent specific architectural events. Different implementations may choose different techniques for creating the same abstract architectural element. Consider the number of ways that one might implement pipes, for example. Indeed, one might even find multiple implementation approaches in the same system. Moreover, for the purposes of architectural discovery, no single architectural style or pattern can be used for all systems. For example, sockets might be used to represent many different types of connectors. Therefore, we need a flexible way to associate different implementation styles with architectural styles.

To address these concerns, we adopted the approach illustrated in Figure 1. Monitored events are first filtered by a Trace Engine to select the subset of system observations that must be considered. The resulting stream of events is then fed to a State Engine. The heart of this recognition engine is a state machine designed to recognize interleaved patterns of runtime events and, when appropriate, to produce a set of architectural operations as outputs. Those

operations are then fed to an Architecture Builder that incrementally creates the architecture, which can then be displayed to a user or processed by architecture analysis tools.

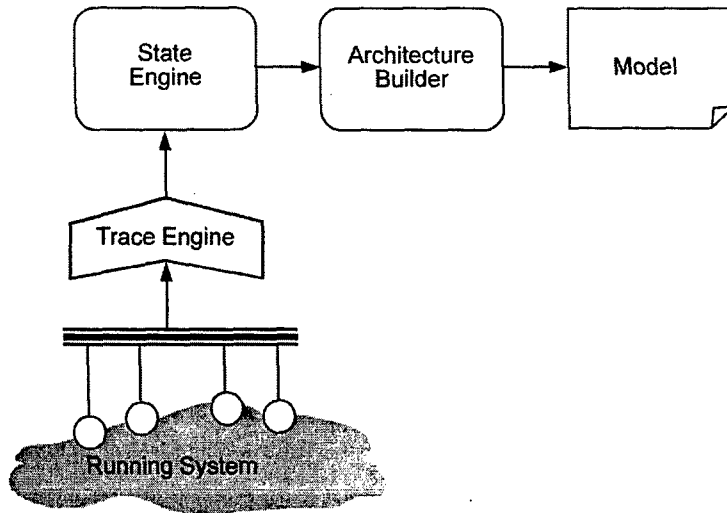


Figure 1: The DiscoTect Architecture

To handle the variability of implementation strategies and possible architectural styles of interest, we provide a language to define new mappings. Given a set of implementation conventions (which we will refer to as an *implementation style*) and a vocabulary of architectural element types and operations (which we will refer to as an *architectural style*), we provide a description that captures the way in which runtime events should be interpreted as operations on elements of the architectural style. Thus, each pair of implementation style and architectural style has its own mapping. A significant consequence is that these mappings can be re-used across programs that are implemented in the same style.

4 DiscoTect State Engine Design

In this section, we discuss the design of the State Engine portion of DiscoTect. We first introduce the language to define the state machine. The semantics for the state machine differ from the standard definition; the informal operational semantics are given in Section 4.1.2. We then illustrate the approach by showing how it can be used to discover the Pipe/Filter architecture of a small Java application. Later (in Section 6), we will present a more substantive example.

4.1 State Machine Definition

To illustrate the definition of state machines, consider a situation in which we want to recognize the creation of instances of some binary connector type. Let's assume the implementation constructs the connector by first creating instances of Read and Write objects through which data are to be communicated. These objects correspond to read and write ports on architectural components. A connector is constructed between those ports when a component invokes the receive method of its Read object, passing it the Write object that contains the data. The state machine to construct this connector will have states that recognize when Read and Write objects are created and when a receive method is called. Transitions between the states will construct elements in the architecture (ports, roles, and connectors).

Complicating this scenario is the fact that the implementation may create Read and Write objects in any order and, in fact, may construct many Read and Write objects before communicating *any* data. This kind of interleaving requires the recognition engine to have multiple active states. Furthermore, because the creation of the connector relies on information from previous states (i.e., the Read and Write objects), we must retain information from previous states for use in evaluations at subsequent states.

A DiscoTect state machine is a graph of states, triggers, actions, and transitions interpreted by the State Engine. States keep track of the progress of architecture discovery. Each state is associated with one or more triggers that define the type of events that can cause transitions between states and that specify the conditions under which these transitions can occur. When a transition is taken, it may produce actions to construct architectures.

4.1.1 Elements of a State Machine

The elements of a state machine are illustrated Figure 2. Specifying a state machine requires the definition of states, triggers, actions, and transitions.

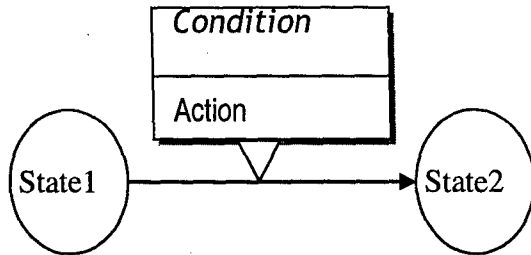


Figure 2: Elements of a State Map

States. States are staging points in the discovery of some architectural action. A state may represent partial knowledge of the architecture—for example, the knowledge that a connector has been created, but we don't yet know which components it connects—and allows us to build complex mappings to combine pieces of information into coherent architectural actions. States are linked by transitions, which form a graph representing implementation flows leading to architectural actions.

Each state in the state machine is associated with a set of state variables. A variable v is present in state s if, for every incoming transition of s , v is either defined on the transition or present in predecessor states. Variables must be defined on every incoming edge to ensure well-defined values. Conditions and actions on the outgoing transitions for s can refer to the variables present in s , as well as any new variables defined by the transition.

Triggers. A trigger consists of two parts: (1) an event specification and (2) a condition that must be met before the transition can occur. In our current prototype, three types of parameterized events can be received from the running system (via the Trace Engine):¹



Call (method, caller, callee): A *Call* event occurs when a method is invoked in the running system. Each *Call* event includes the name of the method, caller, and callee.



Init (constructor, creator, instance): An *Init* event occurs when a constructor is invoked to instantiate a new object. The event contains the name of the constructor, the name of the element requesting the constructor (in the *creator* parameter), and the name and type, collected in the *instance* parameter, of the new element.



Modify (owner, field, value): A *Modify* event occurs when a member variable of an object is assigned a value. The event includes the name of the owner object of the field, the name of the field, and the value that was assigned to the field.

¹ The icons next to each listed item show how the event types are indicated in figures containing state machines. Although our current implementation uses only three types of events associated with object-oriented implementations, the approach could easily accommodate other events and programming styles.

When a state is activated by an event, the parameters of that event are recorded as state variables that can be referred to by subsequent state trigger conditions or actions. In this way, an architectural action can use information from previous states. (We will illustrate how to access these state variables below.)

Conditions are written as Boolean expressions over values of state variables (derived from parameters of the current event or the events of previous states). Conditions may also use operators to build up more complex expressions. For example, the expression `v1 == v2` returns true if `v1` is equal to `v2`, and `v1 contains "read"` returns true if `v1` contains the string "read."

To illustrate, consider a trigger that contains a Modify (✎) event and the following condition:

```
field contains "Reader.lock" && owner == S3.instance
```

This condition is true when the field parameter of the Modify event contains the string "Reader.lock" and the owner parameter is equal to the instance parameter for the Init event that activated S3. (S3.instance is an example of accessing a state variable that was recorded earlier.)

Actions. An action specifies a sequence of architecture-related operations that create or modify the software architecture of the running system. Actions are linked directly to the style of the target architecture and are expressed using operations appropriate to that architectural style. For example, a pipe/filter style might include operations for creating pipes and filters, and a client-server style might include operations for creating and connecting clients to servers. Similar to event parameters, operations may explicitly define values of state variables through assignment, so these values may be used in later actions and conditions.

4.1.2 Informal Operational Semantics

DiscoTect must deal with sequences of events that are interleaved. To do this, DiscoTect may maintain more than one concurrently active state in a state machine. Each active state is called a *state activation*. Each activation consists of a state and a binding for all variables in that state. DiscoTect provides three forms of transitions: (1) *ordinary*, (2) *fork*, and (3) *join*. Like other state machines, ordinary transitions remove one state activation and add another. To support concurrency, DiscoTect also supports *fork* transitions that leave the original state activations in place while also creating a new state activation in parallel with the original. Likewise, DiscoTect has a *join* transition that merges two or more source state activations into a single destination activation.

The current state of the State Engine is a set of state activations. The state engine begins with a single activation for the initial state in the state machine. Whenever an event is received from the trace engine, it is matched against all outgoing transitions from all current state activations. If the event matches the event specification for one or more transitions and the condition for the transition evaluates to true, each matching transition is taken.

For ordinary transitions (i.e., non-forking), the source activation is removed, and the new activation is added for the destination state. Variables in the new state are bound to values defined in the transition or, if they are not defined there, to the values of the corresponding variables in the source activation.

If the transition is a *fork*, the machine retains the source state activation while creating the destination activation. If the transition is a *join*, it can be triggered only if a state activation is present for all of the source states of the *join*. In this case, the source activations are removed, and the destination activation is created as usual.

Consider the state machine fragment in Figure 3,² and assume that S1 is currently active. When S2 becomes active (because the trigger on the transition into S2 is fired), S2's activation consists of the following state variables:

- instance, creator, and constructor from the *Init* event of the trigger
- filter, which is the result of an operation in the action
- S1.method, S1.caller, and S1.callee, which are copied variables from state S1

These variables may be referred to later as S2.instance, S2.filter, and so forth

The transition from S1 to S2 is a *fork* transition. When it occurs, the state activation for S1 is retained and a new state activation for S2 is spawned. This forking allows the creation of other filter components to be tracked by the original state activation for S1, while allowing the new state activation for S2 to track subsequent events happening to the filter created by the transition. In this way, the state machine can keep track of interleaved architectural mappings.

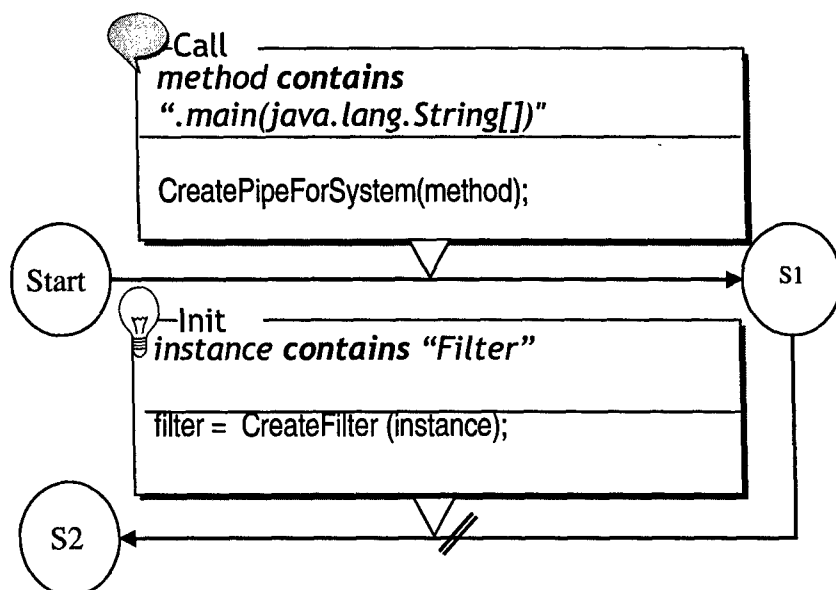


Figure 3: State Machine for Discovering Filter Components

² Throughout this report, we denote a *fork* transition by adding the // icon on the transition.

4.2 Pipe/Filter Example

To illustrate the use of DiscoTect for discovering an architectural model, consider a simple example in a Pipe/Filter architectural style. Assume that the style defines three component types: a type each for data input and output files (called *InFile* and *OutFile*), and a Filter type whose instances consume inputs and produce outputs. There is also a Pipe connector type and interface types specifying the input and output interfaces of filters and pipes.

Furthermore, assume that the Pipe/Filter style defines the following operators to create elements of the above types:

- CreatePipeFilterSystem (name)
- CreateFilter (name)
- CreatePipe (name)
- CreateReadPort (name, component)
- CreateWritePort (name, component)
- CreateSink (name, pipe)
- CreateSource (name, pipe)
- CreateAttachment (port, role)

For this example, assume that the implementation style uses the following conventions: (1) an instance of any class that has “Filter” in its name represents the construction of a Filter component; and (2) Java *PipedReader* and *PipedWriter* instances are used by filters to communicate data. After the write method of a *PipedWriter* is called and the read method of a *PipedReader* is called, we need to wait for a call to the receive method of the *PipedWriter* before we have all the information to create a Pipe in the architecture (the receive method pairs instances of *PipedReader* and *PipedWriter*, defining the ends of the Pipe).

Knowing the style of the implementation and the style of the architecture, we construct a state machine that represents the mapping described above to allow us to recognize when to construct architectural elements. This state machine can be used to discover the Pipe/Filter architecture of any system adopting these implementation conventions.

As an example, we wrote a system (called PrereqCheck) that is implemented using the conventions described above. It creates a configuration of filters to check that students have fulfilled prerequisites for preregistered courses by taking a stream of student entries from a file, splitting the stream depending on whether prerequisites have been satisfied, checking that students have taken particular courses, and then merging the stream to an output file. The code consists of the following application-specific classes:³

³ The filters in the following list are implemented as classes.

- **SplitFilter** – This filter reads an input file one student entry at a time and determines whether the student is in the computer science (CS) program. If so, the entry is sent to one of the output pipes; if not, the entry is sent to the other pipe.
- **PassFilter** – This filter checks each entry to see if a student has taken a prescribed course, in which case the entry is passed on. Otherwise, the entry is discarded.
- **MergeFilter** – This filter takes two inputs and merges them into one output stream.
- **RegSys** – The RegSys class instantiates and starts the filters. Users can execute this class by providing the input and output file names.

In the remainder of this section, we divide the state machine into two parts—(1) Creating Filters and (2) Connecting Filters with Pipes—which are described below.

Creating Filters. This part is responsible for creating the system and the filters in it. The portion of the state machine for this part is shown in Figure 3. When a *Call* event is received from the Trace Engine, it is matched against the triggers outflowing from all active states. Initially, there is one state activation for the Start state. The State Engine will evaluate the condition on the arc out of the Start state. The transition from Start to S1 in Figure 3 looks for a method name containing the string “.main(java.lang.String[])”; if this condition is satisfied by the *Call* event, the Start activation goes away, S1 becomes active, and the accompanying action is executed. This action creates an empty architectural model of the Pipe/Filter style. After S1 becomes active, the trigger condition is evaluated for all newly intercepted object initializations. In Figure 3, if the *instance* parameter to the *Init* event is a Filter, a new state activation for S2 is forked due to the *fork* transition, and an architectural filter component is constructed by the action. The action parameters indicate that the component name should be captured from the new instance, and the component type is decided by the initialization constructor. This new component is assigned to the state variable Filter so that it can be referenced later (for example, in Figure 4). If we follow through this state machine as above, we obtain two state activations for states S1 and S2, respectively. If a later *Init* event satisfies the filter condition on the outbound arc of S1, another filter component is created, along with another concurrent state activation for S2 (which will have different variable bindings from the first activation)

Running PrereqCheck with just this state machine produces the architecture depicted in Figure 5. Four filters are created, one by the constructor for the SplitFilter class, one by the constructor for MergeFilter, and the other two by the constructor for PassFilter. We use an ID generator to label the architectural counterpart of the runtime object to avoid naming conflicts when multiple instances of the same type exist (for instance, two *PassFilters* in this example).

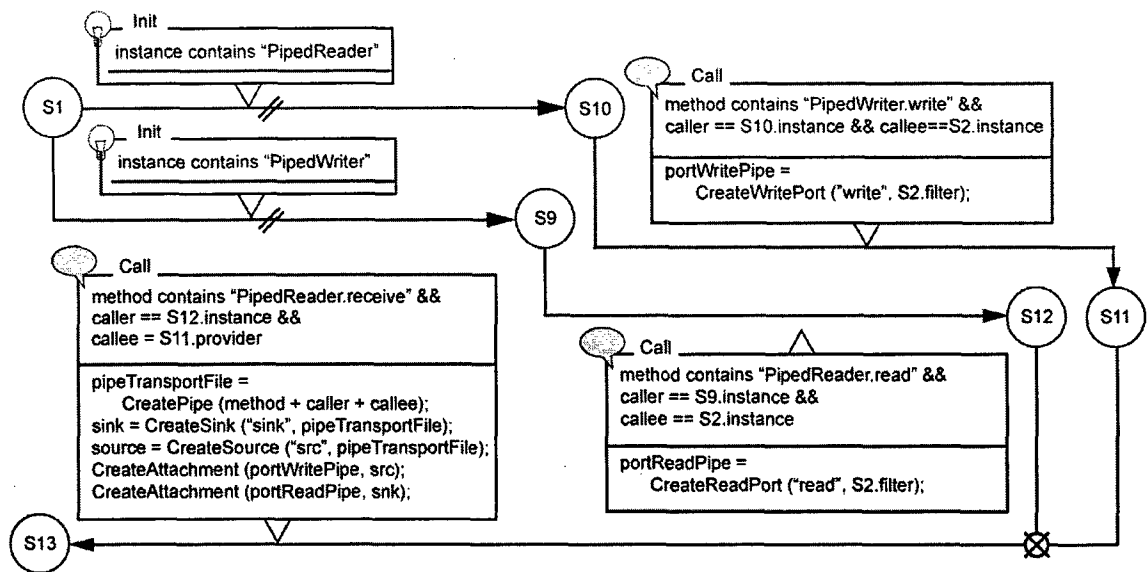


Figure 4: The State Machine Fragments for Discovering Pipe Connections

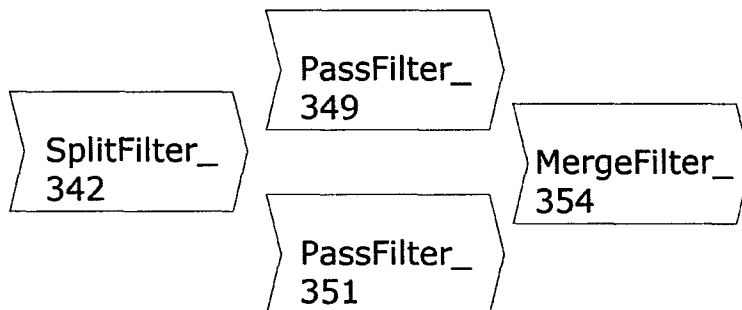


Figure 5: The Architecture Fragment Resulting from Running the System and Using the State Machine Shown in Figure 3

Connecting Filters with Pipes. Recall that the target system uses *PipedReaders* and *PipedWriters* to channel the output from one filter into the input of another. The state machine first creates the ports on filters. For example, a write port is created after noticing the creation of a *PipedWriter* and associating it with an architectural filter when an implementation filter writes to it. Similarly, a read port is constructed when a *PipedReader* is created and a filter reads from it. A pipe is created and connected after calling *PipedReader*'s receive method.

The state machine that performs this process is shown in Figure 4. Newly created *PipedReader* and *PipedWriter* objects are stored by S9/S10 in state variables that can be referred to using *S9.instance* and *S10.instance*. Since the creator is not necessarily the user of those *PipedReader* and *PipedWriter* objects, it is still unclear which filters they belong to, so no port creation action is produced at this point. The filters that are connected by this pipe become apparent only when they are used. When *PipedReader.read* or *PipedWriter.write* is called, the previously recorded *PipedReader* or *PipedWriter* is mapped to ports of the com-

ponents that correspond to the callers. Pipe data flow is signified by calling the receive method of *PipedReader*. This method triggers the *join* transition from S11 and S12 to S13. In this transition, the source state activations are removed, a new state activation for S13 is created, and an action constructs and attaches a pipe between the previously defined *ReadPipe* and *WritePipe* ports.

Putting it all together. The fragments of the state machine from the figures in this section (including one for file output, which is not shown) produce a complete state machine that can discover the architecture of PrereqCheck.

Figure 6 lists the events obtained when running PrereqCheck. This list contains only the events that trigger actions in the state machine (4,550 events are actually received by DiscoTect from the Trace Engine), and for the sake of brevity, we have also removed multiple calls to read and write pipes. The Component Creation part of Figure 6 contains the events that cause the creation of the system and filters by the state machine fragment in Figure 3.

An example of interleaving occurs in the Connection section of the trace. First, the *PipedReaders* and *PipedWriters* are created; then the process of writing to and reading from them begins. So, the pipes are not created sequentially. The State Engine keeps track of separate activations for each pipe, so, in this trace, separate activations after S1 in the state machine (shown in Figure 4) track a *PipedReader/PipedWriter* pair.

After the PrereqCheck system has run, the entire architecture for that run will exist. The resulting architecture from the trace in Figure 6, following the state machine in this section, is shown in Figure 7.

-
1. Call(method="v1.RegSys.main(java.lang.String[])", requestor=null, provider=null)
 2. Init(constructor="v1.SplitFilter", creator=null, instance="v1.SplitFilter(name=", id=342)")
 3. Init("v1.PassFilter", null, "v1.PassFilter(name=", id=349)")
 4. Init("v1.PassFilter", null, "v1.PassFilter(name=", id=351)")
 5. Init("v1.MergeFilter", null, "v1.MergeFilter(name=", id=354)")
 6. Init("java.io.FileReader", "v1.SplitFilter(id=342)", "java.io.FileReader(id=369)")
 7. Init("java.io.BufferedReader", "v1.SplitFilter(id=342)", "java.io.BufferedReader(id=418)")
 8. Init("java.io.FileWriter", "v1.MergeFilter(id=354)", "java.io.FileWriter(id=357)")
 9. Modify(name="java.io.Reader.lock", value="java.io.FileReader(id=369)")
 10. Call("java.io.BufferedReader.readLine()", "v1.SplitFilter(id=342)", "java.io.BufferedReader(id=418)")
 11. Init("java.io.PipedReader", null, "java.io.PipedReader(id=331)")
 12. Init("java.io.PipedReader", null, "java.io.PipedReader(id=334)")
 13. Init("java.io.PipedReader", null, "java.io.PipedReader(id=336)")
 14. Init("java.io.PipedReader", null, "java.io.PipedReader(id=338)")
 15. Init("java.io.PipedWriter", null, "java.io.PipedWriter(id=328)")
 16. Init("java.io.PipedWriter", null, "java.io.PipedWriter(id=329)")
 17. Init("java.io.PipedWriter", null, "java.io.PipedWriter(id=333)")
 18. Init("java.io.PipedWriter", null, "java.io.PipedWriter(id=340)")
 19. Call("java.io.PipedWriter.write(...)", "v1.SplitFilter(id=342)", "java.io.PipedWriter(id=328)")
 20. Call("java.io.PipedWriter.write(...)", "v1.SplitFilter(id=342)", "java.io.PipedWriter(id=329)")
 21. Call("java.io.PipedReader.read()", "v1.PassFilter(id=351)", "java.io.PipedReader(id=338)")
 22. Call("java.io.PipedReader.read()", "v1.PassFilter(id=349)", "java.io.PipedReader(id=331)")
 23. Call("java.io.PipedWriter.write(...)", "v1.PassFilter(id=349)", "java.io.PipedWriter(id=333)")
 24. Call("java.io.PipedWriter.write(...)", "v1.PassFilter(id=351)", "java.io.PipedWriter(id=340)")
 25. Call("java.io.PipedReader.read()", "v1.MergeFilter(id=354)", "java.io.PipedReader(id=334)")
 26. Call("java.io.PipedReader.read()", "v1.MergeFilter(id=354)", "java.io.PipedReader(id=336)")
- Component Creation** (Steps 1-5)
- File Input** (Steps 6-9)
- Connection** (Steps 10-24)
- File Output** (Steps 25-26)

Figure 6: Relevant Output from the Event Filter

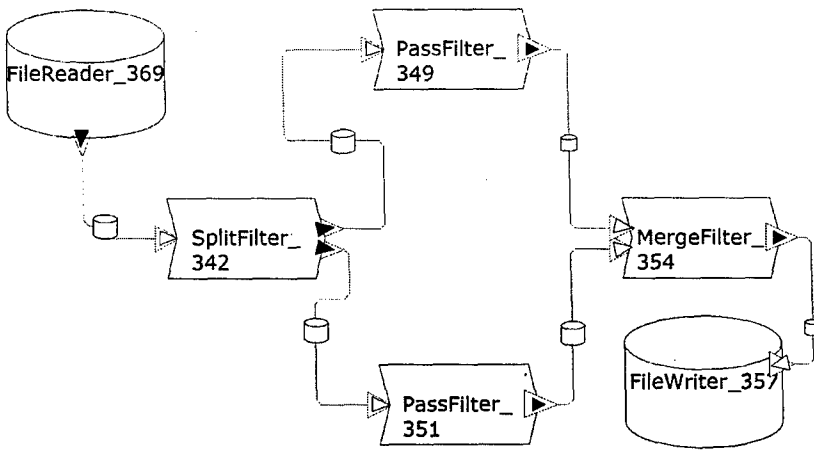


Figure 7: The Discovered Architectural Model of PrereqCheck

5 Implementing DiscoTect

Recall from Section 3 that we need to solve three challenges in order to provide a general framework for discovering architectures. In this section, we discuss our implementation for each of these challenges.

Monitoring: The Trace Engine uses the Java Platform Debugger Architecture (JPDA) to capture system runtime events. The JPDA provides a communication channel between a debugger and a target system. The debugger can send requests to the host virtual machine of the target system querying for certain types of events. The host virtual machine can dispatch events to denote changes of state in the target system. The Trace Engine acts in the role of debugger and sends requests to the virtual machine(s) hosting the target system; the Trace Engine queries for three types of events: (1) object instantiations, (2) method calls, and (3) field modifications. The request also contains a filter that defines the set of classes in which the Trace Engine is interested. At runtime, the target system's virtual machine intercepts requested events generated by any of the classes defined in the filter, queues each event, and sends it to the Trace Engine. Upon receiving a runtime event, the Trace Engine classifies it; converts it into an *Init*, *Call*, or *Modify* event; and puts it in the pipe connected with the Logic Engine.

Mapping: The implementation of the DiscoTect State Engine follows the design in Section 4. During initialization, the State Engine parses the state machine definition and activates the initial state. Then, it keeps scanning the stream sent from the Trace Engine and evaluating the newly produced events with the trigger conditions of currently active states. If a trigger condition out of an active state is satisfied, the target state is activated, and any associated architectural actions are fired.

Architecture Building: We represent architectures using the Acme architecture description language [Garlan 00]. (However, we are not restricted to this language; in principle, any architecture language could serve in this capacity.) Operations on Acme architectures are defined in a library that provides the operations that form building blocks of architectural actions.

6 Adaptive Architectures for Mobile Systems (AAMS) Case Study

In this section, we present a case study to determine the runtime architecture of AAMS, a simulation testbed for experimenting with mobile system architectural design decisions [Kazman 03]. The testbed allows users to specify usable system resources, tasks, and scheduling strategies and simulates the running of the mobile system. We chose the AAMS testbed because it represents a fairly complex real-world application (approximately 28 KLOC [thousand lines of code]), and the runtime architectural view of the system is well documented. By analyzing AAMS with DiscoTect, we were able to compare our discovery results with the original architecture documentation. This comparison illustrates the use of applying our technique to discover deviations between the architecture discovered by DiscoTect and the documented design architecture of AAMS. Furthermore, we can use this case study to illustrate how we developed and refined the state machines to produce the final architecture.

Figure 8 shows the (informal) runtime architecture of AAMS as presented by Kazman and colleagues [Kazman 03]; the following description of the runtime is also based on the description in this report. The simulation controller forms a simulation from the resources and tasks, their configuration, user activities and events, and information that it reads from a set of configuration and script files. The simulation controller also takes commands from the simulation graphical user interface (GUI) to control runtime parameters and feedback. It then processes each simulation frame to generate the actual performance of the system. Each component in the application and resource layers simulates its own operation. A set of services for file input/output (I/O), error reporting, and logging are available via publish/subscribe services to any simulated object.

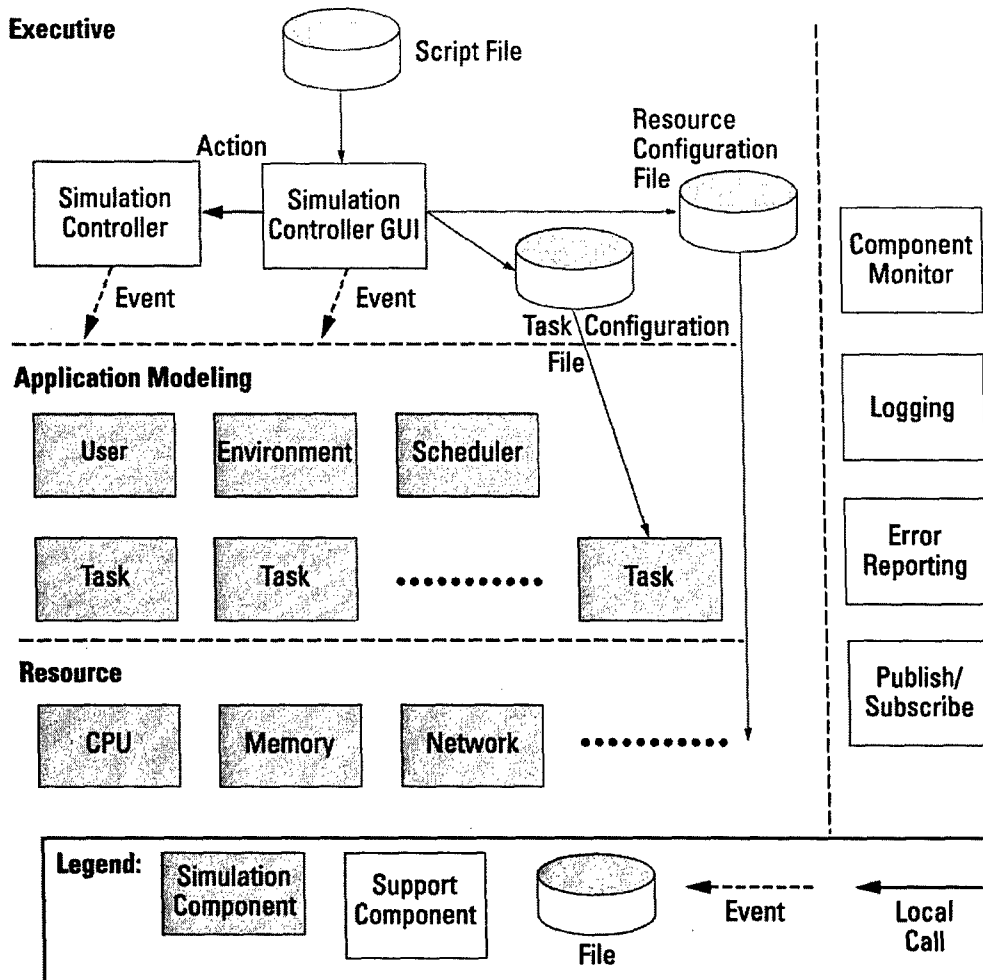


Figure 8: Documented Runtime View of AAMS

6.1 Design of the AAMS State Machine

In this section, we present the steps taken to produce the DiscoText state machine for the purpose of discovering the AAMS architecture model. Typically, writing these state machines is a process of starting with fairly generic state machines to discover components and connections and then refining these state machines to produce architectures that correspond to a particular style. For this case study, we used a specialization of a publish/subscribe style that distinguishes participating components as tasks, resources, and so forth. These extra component types are based on the description of AAMS [Kazman 03].

To develop the final state machine, we first produced a state machine that merely observed object creation and interaction (through procedure calls). We then refined this state machine to classify objects into their architectural counterparts (e.g., resource, task). We also reused the file I/O part from the pipe/filter example.

Up to this point, we had not discovered anything about the publish/subscribe part of the architecture. The preliminary discovery results informed us that all the resource and task components interact with an object of the *PubSub* class using two procedure calls named *publish*

and *subscribe*. We conjectured that the system implements the publish/subscribe process by creating a *PubSub* object and invoking its two methods. This led us to design a state machine for this portion of the architecture. This state machine creates an *EventBus* connector when it notices the instantiation of a *PubSub* object in the implementation. Next, an *EventTaker* role is created when DiscoTect notices two things: (1) a call to the *publish* method of the *PubSub* object and (2) a *Publish* port on the component that corresponds to that call. Then DiscoTect attaches the call and the port. Similarly *PubSub.subscribe* leads to the creation of three things: (1) an *EventSender* role on the *EventBus* providing the method, (2) a *Subscribe* port in the component requesting the method, and (3) the attachment.

6.2 The Discovered Architecture

Applying the above state machine to a running instance of AAMS yields the architectural model shown in Figure 9. We have laid out this model to enable easier comparison with the view in Figure 8. We uncovered four types of discrepancies between the documented architectural view and our discovered one:

1. isolated, extraneous components/connectors. The result shows two *EventBus* connectors, one of which is isolated from the other parts of the system. It indicates that one instance is instantiated but never used. Code optimization should resolve this discrepancy.
2. additional connections between components. Figure 8 does not show any connections between the controller component and simulation components such as tasks and schedulers. Nor does it inform us that some of the support components (Logger and Reporting) also subscribe to the event bus. Ignoring those “backdoor” connections makes the architectural view less accurate; moreover, it might compromise architectural analysis where all meaningful interactions between components should be considered. For example, in evaluating the performance of a publish/subscribe infrastructure, the existence of hidden communication channels could invalidate deadlock analysis.
3. misplaced connections between components. The discovered architecture shows a very different file I/O scheme: instead of the GUI reading three files (see Figure 8), the controller reads two files.
4. missing components/connectors. Two of the components (User and Environment) recorded in the document do not show up in the architecture.

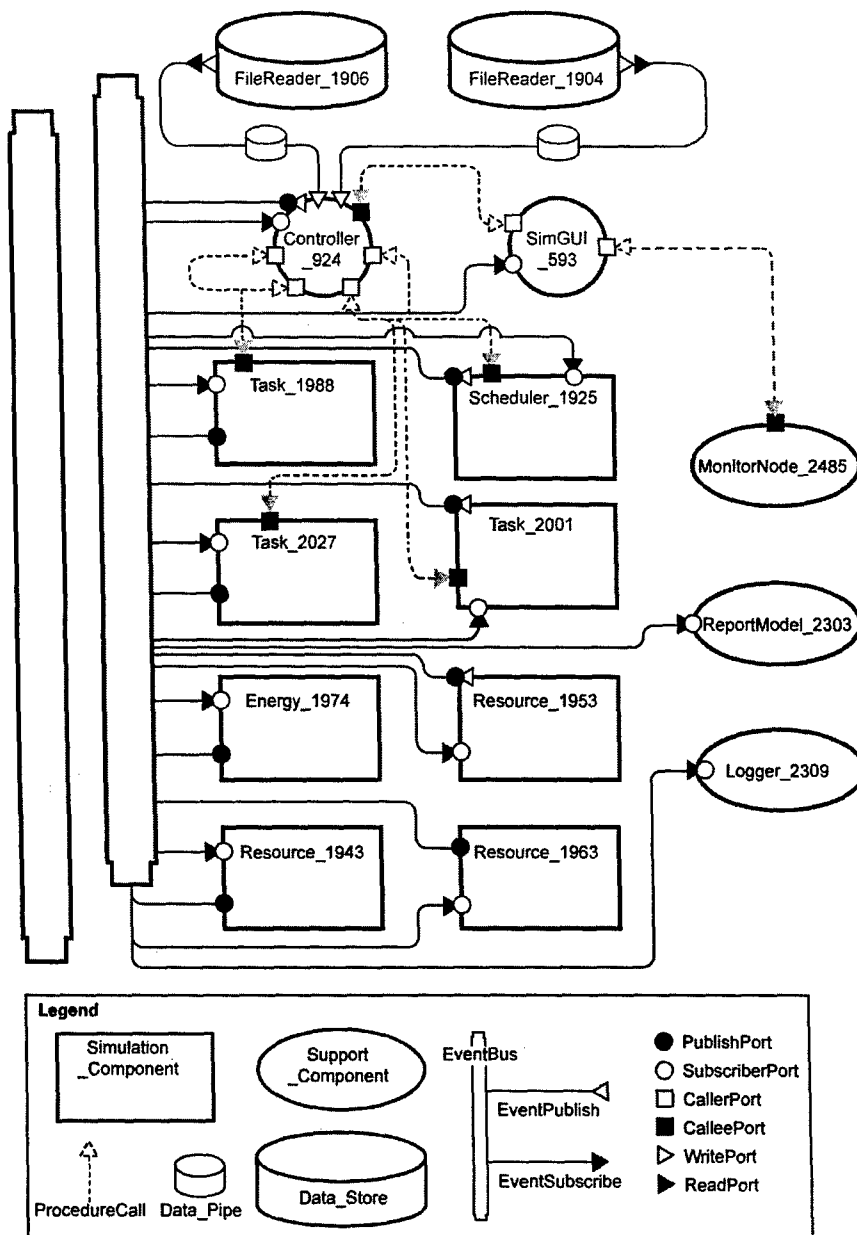


Figure 9: Discovered Architecture of AAMS

To confirm that DiscoTest discovered the actual architecture of the implementation and to understand the discrepancies, we conferred with the AAMS developers. They agreed that DiscoTest produced a more complete and correct architectural description than their diagram and uncovered some errors in their coding. However, the User and Environment components are missing because they represent user interaction and are not actual components in the implementation.

7 EJB Case Study

In this section, we present a second case study to determine the runtime architecture of Duke's Bank Application—a simple Enterprise JavaBeans (EJB) banking application created by Sun Microsystems as a demonstration of EJB functionality. Duke's Bank allows bank customers to access their account information and transfer balances from one account to another. It also provides an administration interface for managing customers and accounts. We use this case study to demonstrate how the architecture of an EJB application can be discovered using DiscoTect. We chose this system because its architecture is well documented in Sun Microsystems' J2EE (Java2 Platform, Enterprise Edition) tutorial [Sun 04], which enables us to compare the actual discovered architecture with the one presented in the documentation.

For this case study, we adopted a new approach—using AspectJ [Eclipse 05, Kiczales 01]—to capture system runtime events. We wrote an aspect that injected advice to object instantiations, method calls, and field modifications. We compiled the Duke's Bank application along with the aspect, using an AspectJ compiler instead of Sun's `javac`, so that system execution events were traced as the application ran. The runtime information was then fed to the Trace Engine, providing the raw material for the recognition process.

Figure 10 gives a high-level view of how the components interact in Duke's Bank system as presented in the J2EE tutorial [Sun 04]. The EJB application has three session beans: (1) `AccountControllerBean`, (2) `CustomerControllerBean`, and (3) `TxControllerBean`. (Tx stands for a business transaction, such as transferring funds.) These session beans provide a client's view of the application's business logic. For each business entity represented in the simplified banking model, the application has a matching entity bean: `AccountBean`, `CustomerBean`, and `TxBean`. The business methods of the `AccountControllerBean` session bean manage the account-customer relationship and get the account information using `AccountBean` and `CustomerBean` entity beans. The `CustomerControllerBean` session bean provides methods for creating, removing, and updating customers through `CustomerBean` entity beans. The `TxControllerBean` session bean handles bank transactions. It accesses `AccountBean` entity beans to verify the account type and to set the new balance, and it accesses the `TxBean` entity bean to keep records of the transactions.

7.1 Design of the EJB State Machine

In this section, we present the steps taken to produce the DiscoTect state machine for the purpose of discovering the Duke's Bank architecture. For this case study, we used a specialization of an EJB style that distinguishes participating components as entity beans, session

beans, bean containers, a database, and so forth. These component types are based on the EJB specification found on the EJB Web site [Sun 05a].

As we did in the previous case study, we first produced a state machine that merely observed object creation and interaction (through procedure calls and object instantiations). We then refined this primitive state machine to classify objects into their architectural counterparts (e.g., beans, bean containers, a database) by checking the class constructor names. For example, we created a *SessionContainer* object when its constructor had the name of "SessionContainer." The relationships between the beans, the bean containers, and the database were captured as follows: According to the EJB specification, the beans are maintained by their corresponding containers, so we connected the beans with the containers controlling them by observing the procedure calls made by the containers to manage the life cycles of the beans. Knowing that database access was implemented using Java Database Connectivity (JDBC) [Sun 05b], we monitored the standard JDBC application program interfaces (APIs) to uncover the connections between the beans and the database; the interactions between the beans were also monitored and represented as connectors linking the beans together.

7.2 The Discovered Architecture

Applying the state machine just described to a running instance of Duke's Bank yields the architectural model in Figure 11. We have organized the layout of the architectural model in Figure 11 to roughly approximate the layout used in the published architecture (shown in Figure 10).

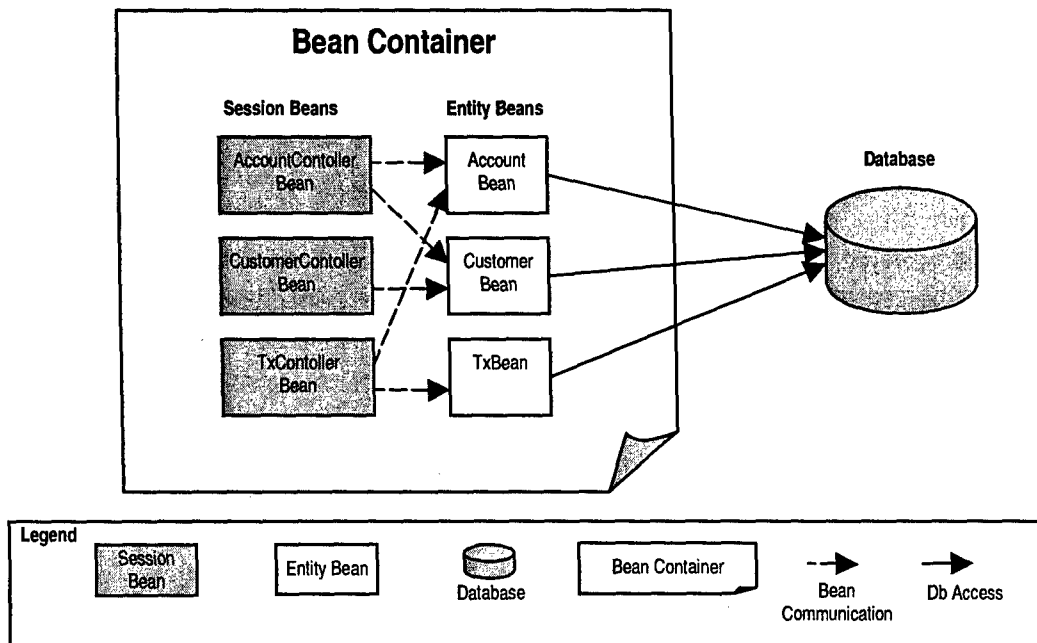


Figure 10: Documented Architectural View of Duke's Bank Application

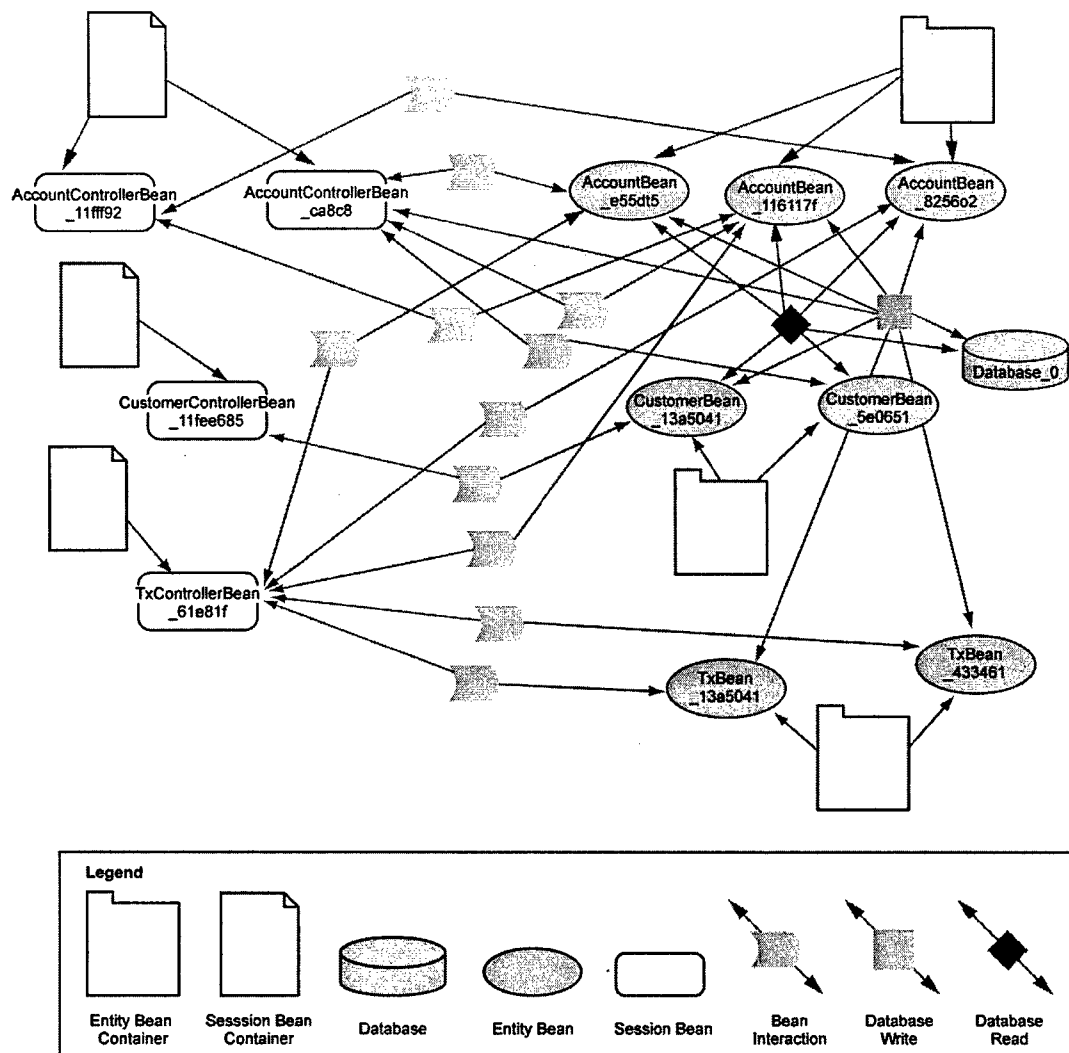


Figure 11: Discovered Architecture of Duke's Bank

We can make the following observations based on our extracted view of the architecture of Duke's Bank:

- reflection of runtime instances. Besides showing the bean and the containers, the discovered result also details each bean and container instance created at runtime. The capacity of tracing the individual bean and container instances is useful for further performance analysis and fault diagnosis. In addition, the relatively complex m to n relationships between beans and bean containers are revealed.
- verification of bean interplay. The interactions between the beans shown in Figure 11 are consistent with those described in the architecture shown in Figure 10: communication channels exist between AccountControllerBean and AccountBean; AccountControllerBean and CustomerBean; CustomerControllerBean and CustomerBean; TxControllerBean and TxBean; and TxControllerBean and AccountBean.

- discrepancies in database access. Figure 10 does not show any connections between the session beans and the database, which implies that all database access goes through the entity beans. However a “database write” connector appeared in the discovered architecture presented in Figure 11. Further source code analysis (performed manually) confirmed that AccountControllerBean does write directly to the database. As discussed in the previous section, identifying communication “backdoor” connections like that one is useful for architectural analysis and for ensuring architectural conformance.

8 Lessons Learned and Future Work

In this report, we described a technique for “discovering” the architecture of a running system, using a set of pattern recognizers that convert monitored system observations into architecturally meaningful events. The key idea is to exploit implementation regularities and knowledge of the architectural style that is being implemented to create a mapping that can be applied to any system that conforms to the implementation conventions and to yield a view in that architectural style.

This approach has several advantages. First, it can be applied to any system that can be monitored at runtime. In our case, we have demonstrated two case studies written in Java, but we have recently experimented successfully with the use of AspectC to extract run-time information from C and C++ programs. In general, any monitoring environment that allows us to capture object creation, method invocation, and instance variable assignment will serve as a sufficient foundation for our runtime monitoring. Monitoring technology for other kinds of implementations and system properties is an active research area (see Section 2) that should continue to provide increasing capabilities in the future that we can use as leverage. Second, by simply substituting one mapping description for another, it is possible to accommodate different implementation conventions for the same architectural style or, conversely, to accommodate different architectural styles for the same implementation conventions. Though not described in this report, we have been able to successfully discover the Pipe/Filter architecture of a system implemented using a different pipe convention.

There are, however, several inherent weaknesses to the approach. The first is that it only works if an implementation obeys regular coding conventions. Completely ad hoc bodies of code are unlikely to benefit from the technique. Second, it only works if one can identify a target architectural style, so the mapping “knows” the output vocabulary. Third, as with any analysis based on runtime observations, it suffers from the problem that you can only analyze what is actually executed. Hence, questions like “Is there *any* execution that might violate a set of style constraints?” cannot be answered directly using this method. Fourth, the recognition engine needs to be created via an iterate-and-test paradigm, and hence the results are somewhat dependent on the skill of the recognizer’s creator. Thus, our techniques are best viewed as one of several technologies that architects must have in their arsenal of architecture-conformance checking tools. We believe that DiscoTect can be effectively combined with tools such as Dali [Kazman 99] or ARMIN [O’Brien 03] to provide complementary kinds of analysis, whereby runtime observations can be combined with statically extracted facts. In this way, we should be able to achieve a more complete and accurate picture of the as-built system.

These potential defects also point the way to future research in this area. First is the area of system monitoring, which was already mentioned. Second is the area of codifying the ways in which architectural styles are implemented. As technology advances, implementation techniques will necessarily change, and it will be important to augment the set of mappings as that happens. Third is the area of architectural coverage metrics, similar to coverage metrics for testing. It would be good, for example, to have some confidence that in running a system with various inputs, we have exercised a sufficiently comprehensive part of the system to know what its architecture is. Fourth, we would like to find a way to make the definition of implementation-architecture mappings more declarative. While the operational definition of state machines as the carrier of those mappings is a good first step, we can imagine more abstract forms of characterization that will be easier to create and analyze. Finally, we are developing tool support for defining state machines.

As mentioned above, our implementation can also be improved. In addition to using better monitoring facilities, our approach could be extended beyond just noticing *Create*, *Init*, and *Modify* events, and use any information that can be gleaned from the runtime system through a probing technology (for example, object destruction or thread information). We plan to provide a mechanism to define these system-level events, so they can be used in state machines.

References

URLs are valid as of the publication date of this document.

- [Aldrich 02] Aldrich, J.; Chambers, C.; & Notkin, D. "ArchJava: Connecting Software Architecture to Implementation," 187-197. *Proceedings of the International Conference on Software Engineering*. Orlando, FL, May 19-25, 2002. New York, NY: Association for Computing Machinery, 2002.
- [Allen 94] Allen, R. & Garlan, D. "Formalizing Architectural Connection," 71-80. *Proceedings of the International Conference on Software Engineering*. Sorrento, Italy, May 16-24, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994.
- [Balzer 99] Balzer, R. M. & Goldman, N. M. "Mediating Connectors," 73-77. *Proceedings of 19th IEEE Conference on Distributed Computing Systems. Workshop on Electronic Commerce and Web-Based Applications*. Austin, TX, May 31-June 4, 1999. Los Alamitos, CA: IEEE Computer Society, 1999.
- [Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second edition*. Boston, MA: Addison-Wesley, 2003.
- [Dias 03] Dias, M. & Richardson, D. "The Role of Event Description on Architecting Dependable Systems," 150-174. *Architecting Dependable Systems* (Lecture Notes in Computer Science, Vol. 2677). Berlin, Germany: Springer-Verlag, 2003.
- [Eclipse 05] Eclipse. *Aspectj project*. <http://eclipse.org/aspectj/>. (URL valid as of March 2005)
- [Ernst 01] Ernst, M. D.; Cockrell, J.; Griswold, W. G.; & Notkin, D. "Dynamically Discovering Likely Program Invariants to Support Program Evolution." *IEEE Transactions on Software Engineering* 27, 2 (February 2001): 99-123.
- [Garlan 00] Garlan, D.; Monroe, R. T.; & Wile, D. "Acme: Architectural Description of Component-Based Systems," 47-68. *Foundations of Component-Based Systems* (Edited by Gary T. Leavens & Murali Sitaraman). New York, NY: Cambridge University Press, 2000.

- [Garlan 02]** Garlan, D.; Kompanek, A. J.; & Cheng, S.-W. "Reconciling the Needs of Architectural Description with Object Modeling Notations." *Science of Computer Programming* 44, 1 (July 2002): 23-49.
- [Garlan 03]** Garlan, D.; Cheng, S.-W. & Schmerl, B. "Increasing System Dependability Through Architecture-Based Self-Repair," 61-89. *Architecting Dependable Systems* (Lecture Notes in Computer Science Vol. 2677). Berlin, Germany: Springer-Verlag, 2003.
- [Jackson 99]** Jackson, D. & Waingold, A. "Lightweight Extraction of Object Models from Bytecode," 194-202. *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles, CA, May 16-22, 1999. New York, NY: Association of Computing Machinery, 1999.
- [Kaiser 03]** Kaiser, G.; Parekh, J.; Gross, P.; & Vetto, G. "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems," 22-30. *Proceedings of the 5th International Active Middleware Workshop*. Seattle, WA, June 25, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.
- [Kazman 99]** Kazman, R. & Carriere, S. J. "Playing Detective: Reconstructing Software Architecture from Available Evidence." *Journal of Automated Software Engineering* 6, 2 (April 1999): 107-138.
- [Kazman 03]** Kazman, R.; Asundi, J.; Kim, J. S.; & Sethananda, B. "A Simulation Testbed for Mobile Adaptive Architectures." *Computer Standards and Interfaces* 25, 3 (June 2003): 291-298.
- [Kiczales 01]** Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; & Griswold, W. G. "An Overview of Aspect J," 327-353. *Proceedings of ECOOP 2001-Object-Oriented Programming, 15th European Conference*. Budapest, Hungary, June 18-22, 2001. Berlin, Germany: Springer-Verlag, 2001.
- [Luckham 97]** Luckham, D. C. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," 329-358. *Proceeding of the DIMACS Partial Order Methods Workshop*. Princeton, NJ, July 24-26, 1996. Providence, RI: American Mathematical Society, 1997.
- [Madhav 96]** Madhav, N. "Testing Ada 95 Programs for Conformance to Rapide Architectures," 123-134. *Proceedings of Reliable Software Technologies - Ada Europe 96*. Montreaux, Switzerland, June 10-14, 1996. Berlin, Germany: Springer-Verlag, 1996.

- [Murphy 95]** Murphy, G. C.; Notkin, D.; & Sullivan, K. J. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," 18-28. *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Washington, DC, October 10-13, 1995. New York, NY: Association for Computing Machinery, 1995.
- [O'Brien 03]** O'Brien, L. & Stoermer, C. *Architecture Reconstruction Case Study* (CMU/SEI-2003-TN-008, ADA413856). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.
<http://www.sei.cmu.edu/publications/documents/03.reports/03tn008.html>
- [Reiss 03]** Reiss, S. "JIVE: Visualizing Java in Action Demonstration Description," 820-821. *Proceedings of the International Conference on Software Engineering*. Portland, OR, May 3-10, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.
- [Shaw 95]** Shaw, M.; Deline, R.; Klein, D.; Ross, T. L.; Young, D. M.; & Zelenik, G. "Abstractions for Software Architecture and Tools to Support Them." *IEEE Transactions on Software Engineering* 21, 4 (April 1995): 314-225.
- [Sun 04]** Sun Microsystems, Inc. *The J2EE Tutorial, Second Edition*.
<http://java.sun.com/docs/books/j2eetutorial/index.html> (2004).
- [Sun 05a]** Sun Microsystems, Inc. *EJB Downloads*.
<http://java.sun.com/products/ejb/docs.html> (2005).
- [Sun 05b]** Sun Microsystems, Inc. *J2EE JDBC Technology*.
<http://java.sun.com/products/jdbc> (2005).
- [Taylor 96]** Taylor, R. N.; Medvidovic, N.; Anderson, K. M.; Whitehead, E. J.; Robbins, J. E.; Nies, K. A.; Oriezy, P.; & Dubrow, D. "A Component- and Message-Based Architectural Style for GUI Software." *IEEE Transactions on Software Engineering* 22, 6 (June 1996): 390-406.
- [Vestal 96]** Vestal, S. *MetaH Programmer's Manual, Version 1.09* (Technical Report). Plymouth, NJ: Honeywell Technology Center, April 1996.
- [Vieira 00]** Vieira, M.; Dias, M.; & Richardson, D. J. "Software Architecture Based on Statechart Semantics," 133-137. *Proceedings of the 10th International Workshop on Software Specification and Design*. San Diego, CA, November 5-7, 2000. Los Alamitos, CA: IEEE Computer Society, 2000.

- [Walker 98]** Walker, R. J.; Murphy, G. C.; Freeman-Benson, B.; Wright, D.; Swanson, D.; & Isaak, J. "Visualizing Dynamic Software System Information Through High-Level Models," 271-283. *Proceedings of OOPSLA'98: Conference on Object-Oriented Programming, Systems, and Applications*. Vancouver, BC, October 18-22, 1998. New York, NY: Association for Computing Machinery, 1998.
- [Walker 01]** Walker, R. J.; Murphy, G. C.; Steinbok, J.; & Robillard, M. P. "Efficient Mapping of Software System Traces to Architectural Views," 31-40. *Proceedings of CASCON 2000* (Edited by S. A. MacKay & J. H. Johnson). Mississauga, Ontario, Canada, November 13-16, 2000. Toronto, Ontario, Canada: IBM Canada, Ltd., 2001.
- [Wells 01]** Wells, D. & Pazandak, P. "Taming Cyber Incognito: Surveying Dynamic/Reconfigurable Software Landscapes." *Proceedings of the 1st Working Conference on Complex and Dynamic Systems Architectures*. Brisbane, Australia, December 12-14, 2001. Brisbane, Australia: Distributed Systems Technology Center at the University of Queensland, 2001.
- [Zeller 01]** Zeller, A. "Animating Data Structures in DDD," 69-78. *Proceedings of the First International Program Visualization Workshop*. Finland, July 7-8, 2000. Porvoo, Finland: University of Joensuu, 2001.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
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 the 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 December 2004	3. REPORT TYPE AND DATES COVERED Final		
4. TITLE AND SUBTITLE Discovering Architectures from Running Systems: Lessons Learned		5. FUNDING NUMBERS F19628-00-C-0003		
6. AUTHOR(s) Hong Yan, Jonathan Aldrich, David Garlan, Rick Kazman, Bradley Schmerl				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2004-TR-016		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPB 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2004-016		
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE		
13. ABSTRACT (MAXIMUM 200 WORDS) One of the challenging problems for software developers is guaranteeing that a system as built is consistent with its architectural design. This report describes a technique that uses automatically generated runtime observations of an executing system to construct an architectural view of the system. In this technique, mappings are developed that exploit regularities in system implementation and architectural style. These mappings describe how low-level system events can be interpreted as more abstract architectural operations. In addition, this report describes the current implementation of a tool, called DiscoTect, that uses these mappings, and it shows how DiscoTect can highlight inconsistencies between implementations and architectures. Furthermore, two case studies are provided that illustrate how DiscoTect works and how it can be applied to real-world systems.				
14. SUBJECT TERMS AAMS, Adaptive Architecture for Mobile Systems, architecture design, DiscoTect, Enterprise JavaBeans, EJB, system architecture		15. NUMBER OF PAGES 42		
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	