

DiscoTect: A System for Discovering Architectures from Running Systems

Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman
Carnegie Mellon University

5000 Forbes Ave, Pittsburgh PA 15213

{yh,garlan,schmerl,aldrich}@cs.cmu.edu, kazman@sei.cmu.edu

Abstract

One of the challenging problems for software developers is guaranteeing that a system as built is consistent with its architectural design. In this paper we describe a technique that uses run time observations about an executing system to construct an architectural view of the system. With this technique we develop mappings 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. We describe the current implementation of a tool that uses these mappings, and show that it can highlight inconsistencies between implementation and architecture.

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 run time components (e.g., clients, servers, databases), their interactions (e.g., RPC, event multicast), 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. 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., [1]) 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 [22,23,24]. While effective when it can be applied, this technique has limited applicability. In particular, it can usually only be applied in situations where engineers are required to use specific

architecture-based development tools, languages, and implementation strategies. For systems that are composed out 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 [12,14,19]. When an implementation is sufficiently constrained that modularization and coding patterns can be identified with architectural elements, this 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 run-time structures that are the essence of most architectural descriptions [8]. In particular, the actual run-time structures may not even be known until the program runs: clients and servers may come and go dynamically; components (e.g., DLLs) not under direct control of the implementers may be dynamically loaded, etc.

A third, relatively unexplored, technique is to determine the architecture of a system by examining its behavior *at run time*. 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, it gives an accurate image of what is actually going on in the real system, it can accommodate systems whose architecture changes dynamically, and it 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 actions. 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 paper 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 one to map implementation styles to architecture styles.

This mapping is defined as a set of conceptually concurrent state machines that are used at run time to track the progress of the system, and output architectural events when predefined run time patterns are recognized. By parameterizing the framework by both architectural and implementation styles, we are able to exploit regularity in systems, while still providing flexibility in defining new abstraction mappings.

In this paper we introduce DiscoTect, a system for discovering the architectures of running object-oriented 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 implementation of DiscoTect in Section 5, and present results from a case study to illustrate the utility of DiscoTect in Section 6. In Section 7 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. These include instrumenting the source code to produce trace information and manipulating runtime artifacts to get the information (e.g., [3] and [28]). There are many technologies available for monitoring systems, and we build on those. However, they do not by themselves solve the hard problem mapping from code to more abstract models. In previous work, we developed an infrastructure doing certain kinds of abstraction [10]; however, this approach was limited to observing properties of a system and reflecting them in an architectural model in a preconstructed architectural model. In this work we show how to create that model in the first place.

Dias et al. [4] use an XML-based language to describe runtime events and use patterns to map these events 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 events has limited expressiveness. For example, it cannot handle the case where the event analyzer initially has interest in one set of events but changes its interest after the interesting 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 [13] uses a collection of temporal state machines to perform pattern matching against runtime events. Our approach is similar, but makes architectural style explicit in the approach.

A number of researchers have investigated the problem of presenting dynamic information to an observer. For example, Reiss [21], Walker [26,27], and Zeller [29]

present information about variables, threads, activations, object interactions, etc. Ernst [5] shows how to dynamically detect program invariants by examining values computed during a program execution, and by looking for patterns and relationships among them. This is somewhat different from detecting architectural structure.

Madhav [18] describes a system that allows Ada 95 programs to be monitored dynamically to check conformance to a Rapide [17] architectural specification. 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 does not rely on architectural construction operations to be embedded in the code.

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., [2,25]). 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: (a) observing a system's runtime behavior, (b) interpreting that runtime behavior in terms of architecturally meaningful events, and (c) representing the resulting architecture. In this paper we are primarily concerned with the second problem of bridging the abstraction gap between system observations and architectural effects.

There are a number of issues that 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 run time infrastructure, initialization of data structures, etc. 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 run time ports of the respective architectural components. This 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. This implies that 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 which 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, there is no single architectural style that can be used for all systems. For example, the use of sockets might be used to represent many different types of connector. This means we need a flexible way to associate different implementation styles with architectural styles.

To address these concerns, we adopt an approach illustrated in Figure 1. Monitored events are first filtered by a Trace Engine to select out 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 output a set of architectural operations. 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.

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* [7]), 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 reused across programs that are implemented in the same style.

4. DiscoTect Design

We now 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 (Section 6) we present a more substantive ex-

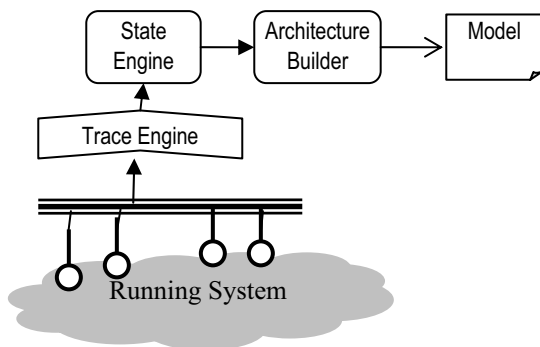


Figure 1. The DiscoTect Architecture.

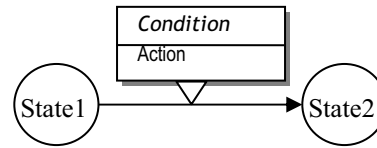


Figure 2. Elements of a state map.

ample.

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 is 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 to 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*, which define the type of events that can cause *transitions* between states, and that specify the conditions under which this can occur. When a transition is taken, it may produce *actions* to construct architectures.

The elements of a state machine are illustrated in Figure 0. Specifying a state machine requires the definition of (1) states, (2) triggers; (3) actions; and (4) transitions.

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 up 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

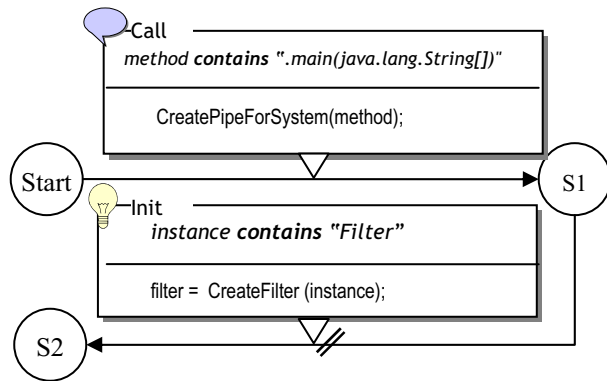





Figure 3. State machine for discovering Filter components.

every incoming transition of s , v is defined either on the transition or is 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: an event *specification*, and a *condition* that must be met for the transition to occur. In our current prototype there are three types of parameterized events that 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.

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

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

¹ The icons next to each list 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 others events and programming styles.

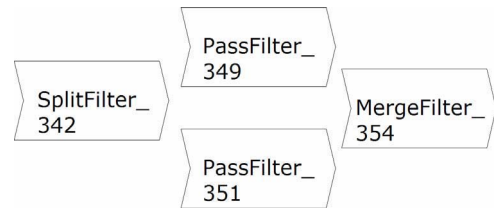



Figure 4. The architecture fragment resulting from running the system and using the state machine Figure 3.

expressions. For example, the expression $v1 == v2$ returns true if $v1$ is equal to $v2$ and $v1$ *contains* "foo" returns true if $v1$ contains the string "foo".

To illustrate, consider a trigger that contains a *Modify* () event and the 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 directly linked to the style of the target architecture, and are expressed using operations appropriate to that architectural style [11]. For example, a pipe-filter style might include operations for creating pipes and filters; a client-server, operations for creating and connecting clients to servers. Similar to event parameters, operations may explicitly define values of state variables through assignment; this is so they may be used in later actions and conditions.

4.1.1 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 activations*. Each activation is a pair of a state and a binding for all variables in that state. DiscoTect provides three forms of transitions: *ordinary*, *fork*, and *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, then 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 not defined there, to the values of the corresponding variables in the source activation.

If the transition is a *fork*, then the machine retains the source state activation while creating the destination activation. If the transition is a *join*, it can only be triggered if there is a state activation 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, and
- *S1.method*, *S1.caller*, and *S1.callee*, which are copied variables from state S1.

These variables may be referred to later on as *S2.instance*, *S2.filter*, etc.

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 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.

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 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 let us assume that the implementation style uses the following conventions: (1) an instance of

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

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 pre-registered 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:

SplitFilter - This filter reads an input file one student entry at a time and determines whether the student is in the CS program or not. 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 the output file names.

In the remainder of this section, we divide the state machine into several parts and present each part in turn.

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 then the Start activation goes away, S1 becomes active and the accompanying action is executed. This action creates an empty architectural model of the *PipeAndFilter* 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* then 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 5). 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, then 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 4. 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).

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 *PipeReader* is created and a filter reads from it. A pipe is created and connected after calling *PipedReader's* *receive* method.

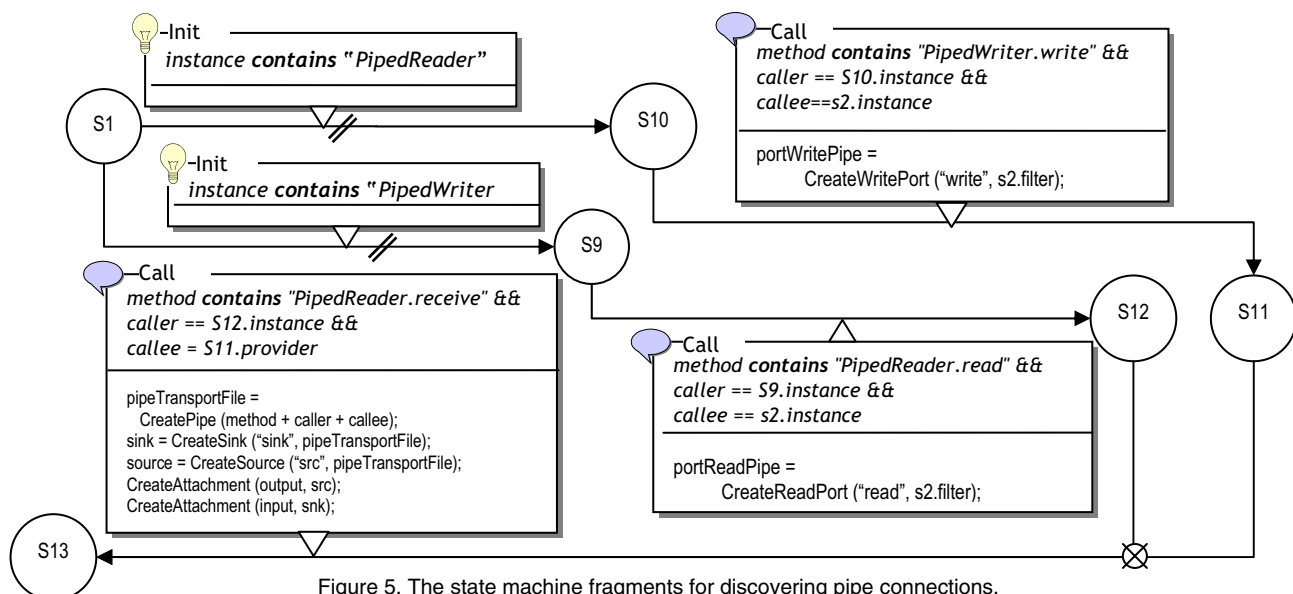
The state machine that achieves this is given in Figure 5. Newly created *PipedReader/PipedWriter* objects are stored by S9/S10 in state variables that can be referred to

using *S9.instance* and *S10.instance*. Since the creator is not necessarily the user of those *PipedReader/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 *PipeWriter.write* is called, the previously recorded *PipedReader/PipeWriter* is mapped to ports of the components 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 is constructed and attached.

Putting it all together. The fragments of the state machine from the figures in this section (including one for file output, 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 (there are actually 4550 events 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 the figure has events causing creation of the system and filters by the state machine in Part 1.

An example of interleaving occurs in the Connection section of the trace. First, the *PipedReaders* and *PipedWriters* are created, then writing to and reading from them commences. So, the pipes are not created sequentially. The State Engine keeps track of separate activations for each of the pipes, so that in this trace there are separate activations after S1 in the state machine in Figure 5, to track a pair of *PipedReader* and *PipedWriter*.



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)")
27. ... more read and write calls
28. Call("java.io.PipedReader.receivedLast()", "java.io.PipedWriter(id=328)", "java.io.PipedReader(id=331)")
29. Call("java.io.PipedReader.receivedLast()", "java.io.PipedWriter(id=329)", "java.io.PipedReader(id=338)")
30. Call("java.io.PipedReader.receivedLast()", "java.io.PipedWriter(id=333)", "java.io.PipedReader(id=334)")
31. Call("java.io.PipedReader.receivedLast()", "java.io.PipedWriter(id=340)", "java.io.PipedReader(id=336)")
32. Call("java.io.Writer.write(java.lang.String)", "v1.MergeFilter(id=354)", "java.io.FileWriter(id=357)")

Figure 6. Relevant output from the event filter.

At the end of running the PrereqCheck system, the entire architecture for that run has been created. The resulting architecture from the trace in Figure 6, following the state machine in this section, is shown in Figure 7.

5. Implementation of DiscoTect

Recall from Section 3 that to provide a general framework for discovering architectures, we need to solve three challenges. 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

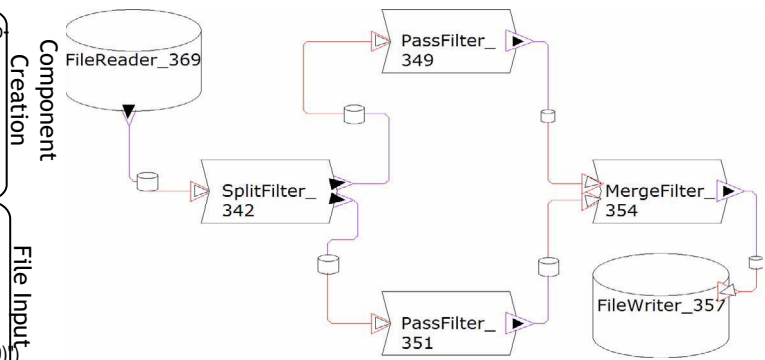


Figure 7. The discovered architectural model of PrereqCheck.

events. 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 querying for three types of events: object instantiations, method calls and field modifications. The request also contains a filter that defines the set of classes the Trace Engine is interested in. At runtime, the target system's virtual machine intercepts requested events generated by any of the classes defined in the filter, queues it, and sends it to the Trace Engine. Upon receiving a runtime event, the Trace Engine classifies it, converts it into one of Init, Call or Modify, 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 [9]. Operations on Acme architectures are defined in a library that provides operations that form building blocks of architectural actions.

6. AAMS Case Study

In this section we present a case study to determine the run time architecture of AAMS, a simulation test-bed for experimenting with mobile system architectural design decisions [15]. The test-bed allows users to specify usable system resources, tasks and scheduling strategies, and simulates the running of the mobile system. We chose AAMS because it represents a fairly complex real world application (approximately 28KLOC), and the runtime architectural view of the system is well documented. This allows us to compare our discovery result with their

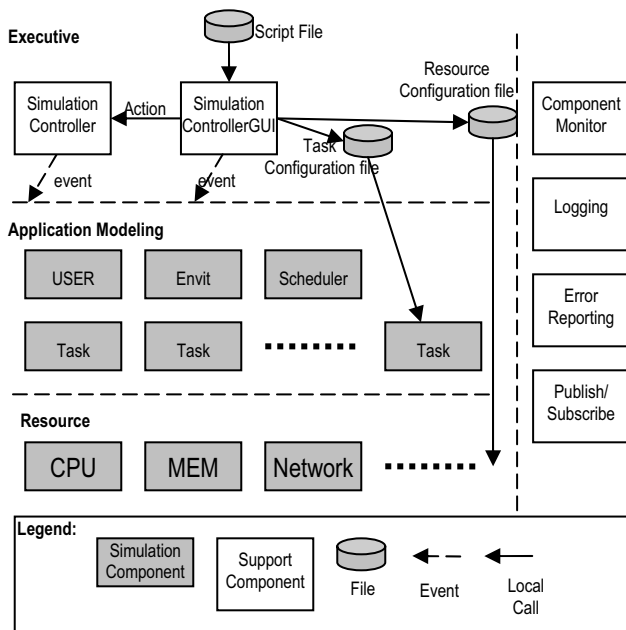


Figure 8. Documented runtime view of AAMS [15].

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 in [15]; the following description of the runtime is also based on the description in this paper. The Simulation Controller forms a simulation from 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 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 I/O, Error Reporting and Logging are available via publish/subscribe to any simulated object.

6.1. Design of AAMS State Machine

In this section we present the steps taken to produce the DiscoTect state machine to discover 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 corresponding to a particular style. For this case study we used a specialization of a publish/subscribe style that distinguishes participating components as tasks, resources, etc. These extra component types are based on the description of AAMS found in [15].

To develop the final state machine, we first produced a state machine that merely observed object creation and interaction (through procedure call). We then refined this to classify objects into their architectural counterparts (e.g., Resource, Task, etc.). We also reused the File IO 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 publish/subscribe 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. Once this has been done, an *EventTaker* role is created when DiscoTect notices a call to the *publish* method of the *PubSub* object, and a *Publish* port on the component corresponding to the call, and attaches them. Similarly *PubSub.subscribe* leads to the creation of an *EventSender* role on the *EventBus* providing the method, the creation of a *Subscribe* port in the component requesting the method, and the creation of the attachment.

6.2. The Discovered Architecture

Applying the above state machine to a running instance of AAMS yields the architectural model in Figure 9. We have laid out this model to enable easier comparison with the view in Figure 8. By comparison with 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 (c.f. Figure 8), the controller reads two files.

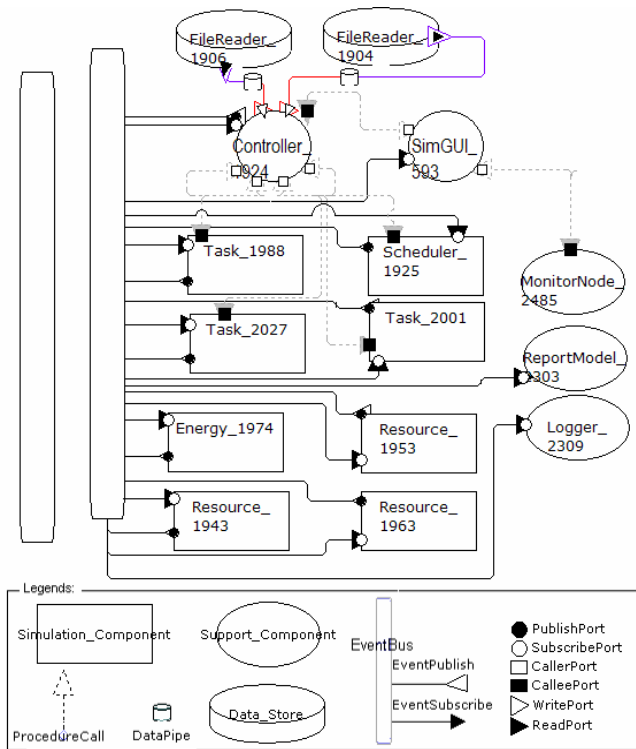


Figure 9. Discovered architecture of AAMS.

4. *Missing components/connectors.* Two of the components (USER and Environment) recorded in the document do not show up in the architecture.

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

7. Discussion and Future Work

In this paper 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 to yield an view in that architectural style.

There are a number of advantages of this approach. First, it can be applied to any system that permits runtime monitoring. In our case, any Java program can be used, since the Java runtime provides built-in facilities for monitoring object creation, method invocation, and instance variable assignment. Our current implementation uses JPDA, which causes a 10X slowdown in the target

system. To address this, we are investigating AspectJ [16] to provide similar monitoring capabilities. Initial results indicate that AspectJ has negligible impact on the speed of the system. Furthermore, 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 leverage. Second, by simply substituting one mapping description for another, it is possible to accommodate different implementation conventions for the same architectural style, or inversely different architectural styles for the same implementation conventions. Though not described in this paper, we have been able to discover the Pipe/Filter architecture of a system implemented using a different pipe convention. Third, the technique can work with a variety of monitoring technologies and architectural representations. Although we used Java and Acme, one could substitute other technologies with relatively minor changes to the recognizer.

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 that 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 directly answered using this method. Thus our techniques are best viewed as one of several technologies that an architect must have in his arsenal of architecture conformance checking tools.

These potential defects also point the way to future research in this area. First, is the area of system monitoring, 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 is not limited to just noticing Create, Init, and Modify events, but could be extended to 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 that they can be used in state machines. To gain further experience with developing the state machines, we are applying DiscoTect to other systems, most significantly the JBoss [6] framework. We anticipate that this will give us measurements of the amount of reuse that we can get by matching architectural and implementation style.

Acknowledgements

The research described in this paper was supported by DARPA, under Grants N66001-99-2-8918 and F30602-00-2-0616, and by an Software Engineering Institute (SEI) Internal R&D Grant.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In Proc. ICSE 2002.

[2] R. Allen, D. Garlan Formalizing Architectural Connection. In Proc. ICSE 1994.

[3] R.M. Balzer and N.M Goldman. Mediating Connectors. Proc. 1999 ICDCD Workshop on Electronic Commerce and Web-Based Applications, 1999.

[4] M. Dias and D. Richardson. The Role of Event Description on Architecting Dependable Systems (extended version from WADS). Lecture Notes in Computer Science - Book on Architecting Dependable Systems (Spring-Verlag), 2003.

[5] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Tans. on Soft. Eng., 27(2), 2001.

[6] M.Fleury and F. Reverbel. The JBoss Extensible Server. Proc. International Middleware Conference, 2003.

[7] D. Garlan, R.J. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design. Proc FSE 94, 1994.

[8] D. Garlan, A.J. Kompanek, S.-W. Cheng. Reconciling the Needs of Architectural Description with Object Modeling Notations. Science of Computer Programming vol. 44, 2001.

[9] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000.

[10] D. Garlan, B. Schmerl, and J. Chang. Using Gauges for Architecture-Based Monitoring and Adaptation. Proc. 1st Working Conference on Complex and Dynamic Systems Architecture, 2001.

[11] D. Garlan, S.-W. Cheng, B.Schmerl. Increasing System Dependability through Architecture-based Self-repair. Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (Eds). LNCS 2677, Springer-Verlag, 2003.

[12] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In Proc. ICSE 1999.

[13] G. Kaiser, J. Parekh, P. Gross, and G. Vetto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. Proc. 5th International Active Middleware Workshop, 2003.

[14] R. Kazman, and S.J. Carriere. Playing Detective: Reconstructing Software Architecture from Available Evidence. Journal of Automated Software Engineering 6(2), 1999

[15] R. Kazman, J. Asundi, J.S. Kim, and B. Sethananda. A Simulation Testbed for Mobile Adaptive Architectures, Computer Standards and Interfaces, to appear, 2003.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of Aspect J. ECOOP 2001.

[17] D.C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. DIMACS Partial Order Methods Workshop, 1996.

[18] N Madhav. Testing Ada 95 Programs for Conformance to Rapide Architecturs. Proc. Reliable Software Technologies –

Ada Europe 96, 1996.

[19] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In Proc. FSE 1995.

[20] G.C. Murphy, D. Notkin. Lightweight lexical source model extraction, ACM TOSEM, 5(3), 1996

[21] S. Reiss. JIVE: Visualizing Java in Action (Demonstration Description). Proc. ICSE 2003.

[22] M. Shaw, R. Deline, D. Klein, T.L. Ross, D.M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. IEEE Trans. on Soft. Eng. 21(4), 1995.

[23] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oriezy, and D. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Trans. on Soft. Eng. 22(6), 1996.

[24] S. Vestel. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, 1996.

[25] M. Vieira, M. Dias, D.J. Richardson. Software Architecture based on Statechart Semantics. Proc. the 10th International Workshop on Component Based Software Engineering, 2001.

[26] R.J. Walker, G.C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. Isaak. Visualizing Dynamic Software System Information through High-level Models. In Proc. OOPSLA'98,

[27] R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard. Efficient Mapping of Software System Traces to Architectural Views. In S.A. MacKay and J.H. Johnson (eds) In Proc. CASCON 2000. .

[28] D. Wells and P. Pazandak. Taming Cyber Incognito: Surveying Dynamic/Reconfigurable Software Landscapes. Proc. 1st Working Conference on Complex and Dynamic Systems Architectures, 2001.

[29] A. Zeller. Animating Data Structures in DDD. Proc. SIGCSE/SIGCUE Program Visualization Workshop, 2000.