Dynamically Discovering Architectures with DiscoTect

Bradley Schmerl
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213 USA
+1 412 268 5889
schmerl@cs.cmu.edu

David Garlan
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213 USA
+1 412 268 5057
garlan@cs.cmu.edu

Hong Yan
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213 USA
yh@cs.cmu.edu

ABSTRACT
One of the challenges for software architects is ensuring that an implemented system faithfully represents its architecture. We describe and demonstrate a tool, called DiscoTect, that addresses this challenge by dynamically monitoring a running system and deriving the software architecture as that system runs. The derivation process is based on mappings that relate low level system-level events to higher-level architectural events. The resulting architecture is then fed into existing architectural design tools so that comparisons can be conducted with the design time architecture and architectural analyses can be re-run to ensure that they are still valid. In addition to the demonstration, we briefly describe the mapping language and formal definition of the language in terms of Colored Petri Nets.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Data Abstraction

General Terms
Measurement, Documentation, Design, Economics, Verification.

Keywords
Architecture discovery, reverse engineering, architecture design tools and analyses.

1. INTRODUCTION
One of the challenges for software architects is ensuring that an implemented system faithfully represents its architecture. The software architecture of a system defines its high-level organization as a collection of interacting components, connectors, and constraints on interaction. Without assurance that the implementation matches the architecture, the value of the architectural design is significantly reduced, since architectural analyses may have little relationship to the deployed system.

One way to address this challenge is to provide tools and techniques that formally relate the software architecture to an implementation. Researchers have proposed various strategies for doing this, including embedding architecture concepts in the source code [1] and conducting static analysis of the code to elicit architectural views [10][16]. However, both approaches are problematic for representing abstract component and connector (C&C) architectural views because C&C views are fundamentally about the runtime layout of a system, which may not be observable in the static artifact.

We have been investigating an approach that circumvents these difficulties by monitoring the system as it runs and relating these runtime observations to a C&C view of the system [20]. In addition to accurately reflecting the runtime architecture of a system, the approach has the benefits that:

- architectural analyses conducted during design can be conducted on the derived runtime architecture to determine whether the system is behaving in the manner intended by its design;
- observations about the run-time architecture can be used by dynamic adaptation tools so that a system may reflect on itself and repair observed problems or mismatches.

In this demonstration, we present a tool, called DiscoTect (Discovering Architectures), which

1. allows mappings to be specified to relate runtime system observations to architectural events that construct C&C architectural views;
2. runs alongside a system, interpreting these mappings to incrementally, and in real time, construct the current implementation architecture; and
3. hooks into existing architectural design tools to present, analyze, and compare the runtime architecture of the system.

2. DISCOTECT OVERVIEW
The DiscoTect architecture is presented in Figure 1. Probes are inserted in the running system to report system-level events such as method calls, object creation, value changes, etc. These *runtime events* are consumed by the DiscoTect engine, which interprets the events according to the specified mapping to produce *architectural events*. Architectural events include component and connector creation, and setting architectural property values. These architectural events are then fed into existing architectural tools to produce the runtime architecture. While this paper focuses on DiscoTect (the area in the dotted box in Figure 1), in the demonstration we will also show how to probe the system and how to view and analyze the architecture.

2.1 TECHNICAL CHALLENGES
Writing mappings between system level and architectural events is challenging for the following reasons:

1. 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

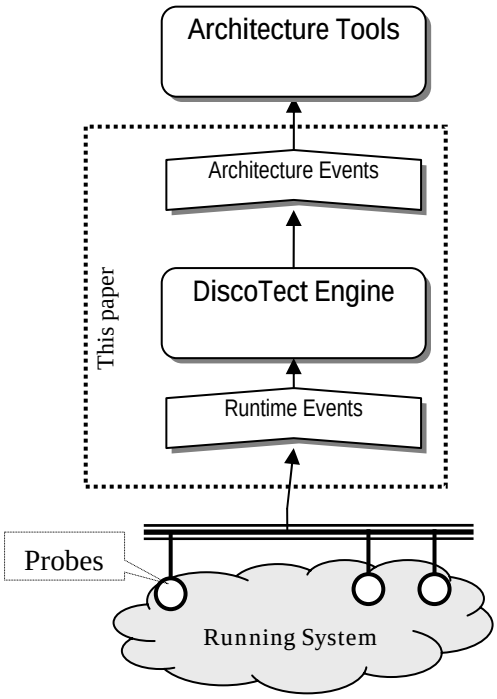

Figure 1. The DiscoTect Architecture

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

2. 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.
3. 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 challenges, we have developed a language, called DiscoSTEP (Discovering Structure Through Event Processing), that is used in DiscoTect to specify the mappings. The execution semantics of DiscoSTEP are specified in terms of Colored Petri Nets.

2.2 DISCOSTEP PROGRAMMING

A DiscoSTEP specification has three main ingredients:

1. **Events.** The types of events produced and consumed by DiscoSTEP are specified in XML Schema definitions. This allows DiscoSTEP to be flexible in the types of events that it receives from a system, and the types of events it passes to the architecture builder. For example, DiscoSTEP could output style-specific events such as the creation of architectural client components in an architectural style. Events are partitioned into those events that are inputs to DiscoSTEP and those that are output by DiscoSTEP. The DiscoSTEP compiler can then conduct some simple type checking to ensure correct type usage.
2. **Rules.** Rules specify how to map a series of system events into architectural events. A rule itself consists of four parts:
    a. **Input events.** Events consumed by the rule.
    b. **Output events.** Events produced by the rule.
    c. **Trigger.** A condition that determines whether inputs match a pattern that will cause the rule to fire.
    d. **Actions.** A set of actions that produce the output events.

```
<init
  type= "java.net.ServerSocket"
  instance_id="0x0f67d9"
/>
```

```
rule CreateServer {
  input { init $e; }
  output { string $server_id;
           create_component $create_server; }
  trigger {?
    contains($e/@type, "ServerSocket")
  ?}
  action = {?
    let $server_id := $e/@instance_id;
    let $create_server :=
      <create_component name="{$server_id}"
                        type="ServerT" />;
  }
}
```

```
<string value="0x0f67d9"/>
<create_component
  name="0x0f67d9" type="ServerT"/>
```

Figure 2. Example Rule with Example Events Consumed and Produced.

Because events are formatted in XML, the triggers and actions are specified using XML Query syntax [19] because it provides a convenient mechanism for manipulating and creating XML elements.

3. **Compositions.** To complete a DiscoSTEP specification, compositions of rules are defined that allow complex sequences of rules to be constructed. Compositions are defined by connecting the outputs from one rule into the inputs of another. Output events that are not passed to another rule are emitted by DiscoTect to construct the architecture.

The center of Figure 2 presents a simple DiscoSTEP rule (called `CreateServer`) for creating a Server component when it notices an `init` event that constructs an implementation object of type `java.net.ServerSocket`. The rule takes one input (an example XML element corresponding to the input is presented at the top of the figure) and two output events (presented at the bottom). The types of input and output events, and the names given to them in the rule, are defined in the input and output section of the rule. The trigger is an XML Query FLOWR expression that checks to see if the value of type attribute of the `init` event contains the string "`ServerSocket`". If it does, then the actions of the rule are fired. The actions are specified in XML Query syntax, and create XML elements for each of the output variables.

Informally, all `init` type XML events will be queued into the rule, but only those matching the trigger will cause the actions to be executed. Rules may specify multiple inputs, in which case each event is queued with the rule, and when sets of these events match a trigger, that set is passed to the action to produce output events. In this way, some order can be given interleaving of architecturally relevant events.

2.3 DISCOTECT FORMAL MODEL

In [20] we presented a preliminary description of the formal model of DiscoTect mappings in terms of state machines. Recently, however, we have redefined the semantics of DiscoSTEP mappings in Colored Petri Nets (CPN) [8]. A full treatment of the formal semantics is obviously not possible in the space of this paper – we refer to the reader to [21] for the full definition. Here, we merely give a flavor of the formal model.

Each rule is modeled as a CPN transition, with the trigger corresponding to a CPN guard and the actions corresponding to CPN arc expressions. Each event type for a rule is modeled as a CPN place, where the color of the place is dictated by the type of the event. The determination of whether an event is an input or output event, and their use in a composition, is used to construct the node function of the CPN that maps places to transitions, and transitions to places. Figure 3 gives an example of the CPN that is formed from the rule in Figure 2. A composition is used to compose transitions and places into a complete CPN. DiscoTect events are modeled as tokens. For each input event that is received by DiscoTect, a token of the color is produced for every input place able to receive that color.

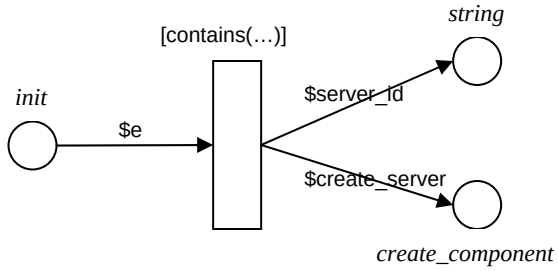

Figure 3. The Colored Petri Net Corresponding to the Rule in Figure 2.

We believe that using Colored Petri Nets is a more natural way of representing the mapping than what we described in [20]. We also use the runtime semantics of CPNs to guide the implementation of the DiscoTect Engine.

3. IMPLEMENTATION

DiscoTect itself is implemented in Java. Once a DiscoSTEP program is compiled, the data structures created to represent the program are serialized and then read in by the DiscoTect Engine for processing. The DiscoTect engine then waits for messages from the probes in a system. To deliver these messages, we use the Java Messaging System (JMS) from Sun.

We use various existing probing technologies to extract monitoring events. In this demonstration, we will illustrate the use of AspectJ [7], to handle low-level monitoring of object creation, method invocation, etc. We provide a library that allows aspects to produce system events formatted as XML that are placed on a JMS event bus to be consumed by DiscoTect.

AcmeStudio [14] is an architecture development environment that is primarily used for constructing architectures at design time. It is implemented as a plugin to the Eclipse environment, a framework for developing integrated development environments. DiscoTect produces architectural events formatted as XML that are forwarded by the AcmeStudio Remote Control plugin, communicating over Java RMI, to incrementally construct the architecture. The analysis capabilities of AcmeStudio can then be used to check the architecture with respect to its style, or conduct analyses such as performance or schedulability.

4. 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 [18]). 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 [6]; 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 [6] 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 [13], Walker [16] [17], and Zeller [22] 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 [12] describes a system that allows Ada 95 programs to be monitored dynamically to check conformance to a Rapide [11] 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.

5. CONCLUSION

In this paper, we have given a brief description of DiscoTect, and how it is used to derive software architectures from runtime observations. The demonstration itself will show the toolset briefly described here, as presented in Appendix A. We have used DiscoTect to derive the architecture of various systems, including EJB systems, a mobile simulation system, and are in the progress of applying it to an automotive infotainment system. Some of case studies are reported in more detail in [20][21]. In all cases, we have discovered discrepancies between the designed architecture and the implemented architecture.

One of the drawbacks of using DiscoTect is that it obviously relies on how much of the system is visited during a particular run. This is similar to the issue of test coverage. For this reason, we see DiscoTect as a complement to existing static analysis techniques. We believe that DiscoTect will be of most use in systems where the run time context is important in ascertaining the architectural implications of an event (such as in distributed systems or systems that use callbacks). We are actively exploring this as future work.

6. ACKNOWLEDGMENTS

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.

7. 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] 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.
[7] IBM. AspectJ Home Page. http://www.aspectj.org.
[8] K. Jensen. Coloured Petri Nets: A High-level Language for System Design and Analysis. In Advances in Petri Nets 1990. LNCS 483, G. Rozenberg (Ed), 1991.
[9] G. Kaiser, J. Parekh, P. Gross, and G. Veletto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. Proc. 5th International Active Middleware Workshop, 2003.
[10] R. Kazman, and S.J. Carriere. Playing Detective: Reconstructing Software Architecture from Available Evidence. Journal of Automated Software Engineering 6(2), 1999
[11] D.C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. DIMACS Partial Order Methods Workshop, 1996.
[12] N Madhav. Testing Ada 95 Programs for Confomance to Rapide Archtiectures. Proc. Reliable Software Technologies – Ada Europe 96, 1996.
[13] S. Reiss. JIVE: Visualizing Java in Action (Demonstration Description). Proc. ICSE 2003.
[14] B. Schmerl, D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. Proc. ICSE 2004.
[15] M. Vieira, M. Dias, D.J. Richardson. Software Architecture based on Statechart Semantics. Proc. the 10th International Workshop on Component Based Software Engineering, 2001.
[16] 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.
[17] 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.
[18] D. Wells and P. Pazandak. Taming Cyber Incognito: Surveying Dynamic/Reconfigurable Software Landscapes. Proc. 1st Working Conference on Complex and Dynamic Systems Architectures, 2001.
[19] XML Query. http://www.w3.org/XML/Query.
[20] H. Yan, D. Garlan, B.Schmerl, J. Aldrich, R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. Proc. ICSE 2004.
[21] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, R. Kazman. Discovering Architectures from Running Systems using Colored Petri Nets. Submitted for publication.
[22] A. Zeller. Animating Data Structures in DDD. Proc. SIGCSE/SIGCUE Program Visualization Workshop, 2000.

APPENDIX A – TOOL AVAILABILITY

DiscoTect will be available for downloading, from www.cs.cmu.edu/~able/discotect. AcmeStudio is currently available from www.acmestudio.org.

APPENDIX B – DEMONSTRATION

In the demonstration, we will walk participants through the full process of instrumenting a system, writing and compiling a DiscoSTEP program that defines the mapping between system level events and architectural events, running DiscoTect alongside a running system to dynamically create the architecture in AcmeStudio, and then using AcmeStudio to check the architecture. The following figures outline this demonstration.

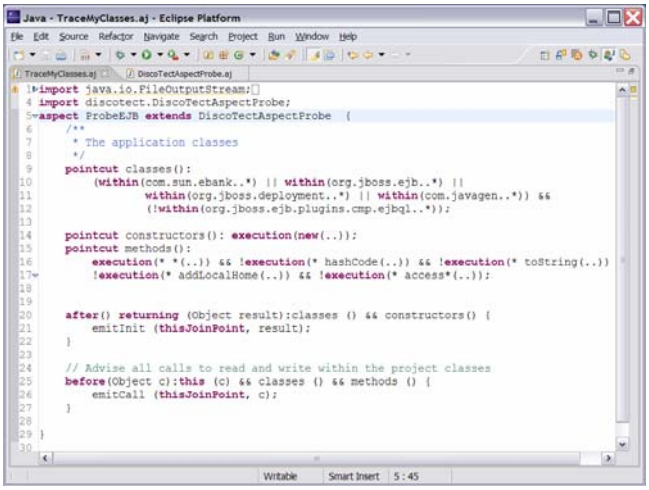

Figure 4. Defining Aspects to Instrument an EJB Application. We provide methods to emit XML events to a JMS bus. The instrumentor needs to determine which methods or constructors need to be probed.

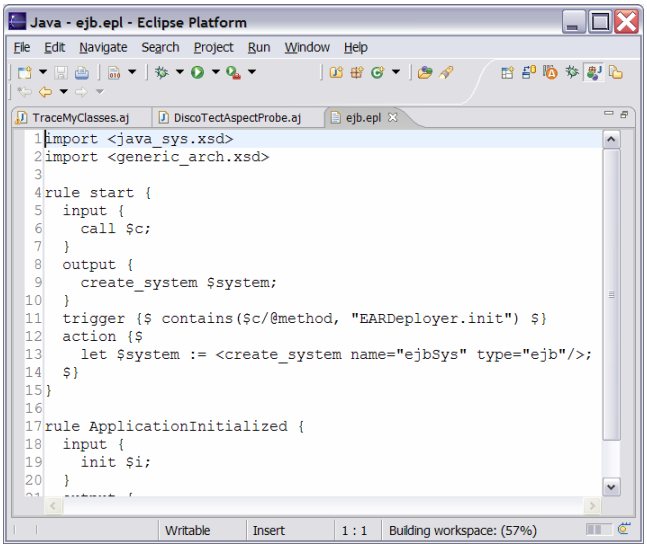

Figure 5. Once the program has been instrumented, we write a DiscoSTEP program to do the mapping. We will demonstrate the support we have for this, which includes compiling the program with feedback from DiscoTect.

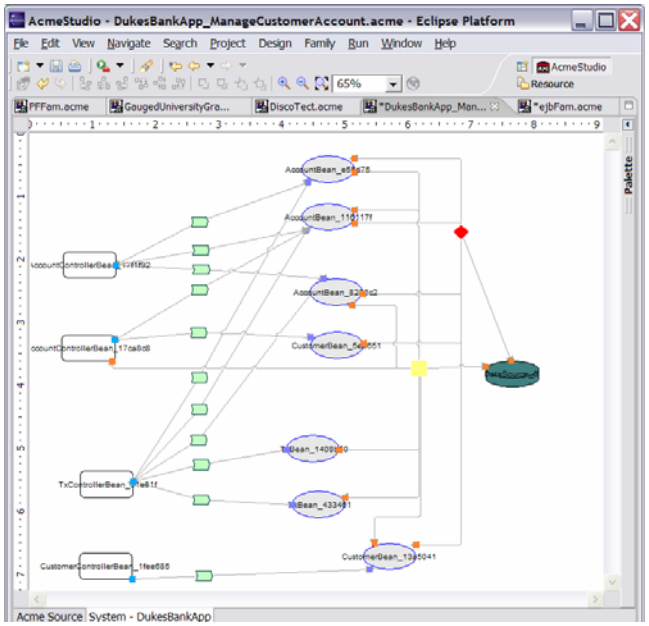

Figure 6. We will run an application and show how the architecture is constructed incrementally. This figure shows the completed EJB architecture of Duke's Bank, a standard EJB example from Sun. The architecture displayed by AcmeStudio.

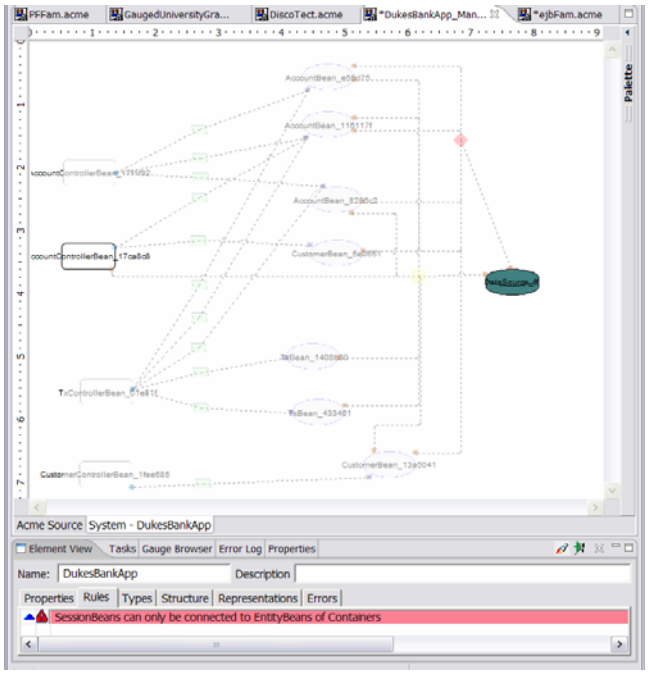

Figure 7. AcmeStudio analysis tools can discover problems in an architecture. In this figure, the EJB architectural style dictates that a Session bean cannot access a Database directly. In the example derived architecture, an AccountController session bean is accessing the database directly. AcmeStudio flags this error (the error is highlighted at the bottom of the figure), and AcmeStudio allows the user to highlight the parts of the architecture caused this error (in the figure, correct parts are dimmed).