Submitted for publication.

Discovering Architectures from Running Systems using Colored Petri Nets

Bradley Schmerl*, Jonathan Aldrich*, David Garlan*, Rick Kazman†, Hong Yan*

School of Computer Science,
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15221
(schmerl|aldrich|garlan|yh)@cs.cmu.edu

Software Engineering Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15221
kazman@sei.cmu.edu

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. In 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, and are formally defined using Colored Petri Nets. In this paper we describe a system, called DiscoTect, that uses these mappings, and we introduce the DiscoSTEP mapping language and its formal definition in terms of Colored Petri Nets. Two case studies showing the application of DiscoTect suggest that the tool is practical to apply to legacy systems and can dynamically verify conformance to a pre-existing architectural specification.

1 Introduction

A well-defined software architecture is critical for the success of complex software systems. 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, event multicast, piped streams), and their properties (e.g., throughputs, latencies, reliabilities) [Bass03, Perry92, Shaw96]. As an abstract representation of a system, an architecture permits many forms of high-level inspection and analysis, allowing the architect to determine if a system's design will satisfy its critical quality attributes. Consequently, over the past decade, considerable research and development

has gone into the development of notations, tools, and methods to support architectural design [Clements 03, Clements01, Medvidovic00].

However, despite considerable progress in developing an engineering basis for software architecture, a persisting difficult problem is determining whether a system as implemented has the architecture as designed. Without some form of consistency guarantees, the validity of any architectural analysis will be suspect, at best, and completely erroneous, at worst.

Currently two general techniques have been used to determine or enforce relationships between a system's software architecture and its 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 [Aldrich02]) 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 [Shaw95, Taylor96, Vestal96].

Ensuring consistency by construction is effective when it can be applied, since tools can guarantee conformance. Unfortunately it 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 [Jackson99, Kazman99, Murphy95]. 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 [Clements03, Garlan02]. In particular, the actual runtime structures may not even be known until the program executes: clients and servers may come and go dynamically; components (e.g., Dynamic Linked Libraries) not under direct control of the implementers may be dynamically loaded; and so forth. Indeed, determining the actual runtime architectural configuration of a system is, in general, undecidable.

A third, relatively unexplored, technique is to determine the architecture of a system by *examining its runtime behavior*. The key idea is that a system's execution can be monitored. Observations about its runtime behavior can then, in principal, be used to infer its dynamic architecture. This approach has the advantage that 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.

However, there are a number of hard technical challenges in making this technique work. The most serious problem 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 paper, we describe a technique, based on a combination of state-based and interaction-based modeling, 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 conceptually as a Colored Petri Net [Jenson94] that are used at runtime to track the progress of the system and output architectural events when predefined runtime patterns are recognized. Thus the mapping provides a kind of behavior modeling, where it is used to identify just those behaviors of a system that are "architecturally significant." One of the important additional features of the approach is the ability to reuse such mappings across different systems. In particular, we exploit regularity in both implementation and architectural styles so that a single mapping can serve as an architectural extractor for a large collection of similar systems, thereby reducing the amortized cost of writing the abstraction mappings, while still providing flexibility.

In the remainder of this paper, we describe the approach in detail, and describing the tool called DiscoTect, that we have implemented. In Section 2, we discuss related work. Section 3 presents the DiscoTect approach, including an overview of the DiscoTect framework, and the DiscoSTEP language used for specifying mappings. We furthermore present a formal semantics for DiscoSTEP that specifies the semantics in terms of Colored Petri Nets. We clarify this with a simple example. In Section 4 we briefly discuss

the implementation of DiscoTect. Sections 5 and 6 describe case studies that show the feasibility of DiscoTect. Finally, in Section 7 we present our conclusions.

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., as described by Balzer and Goldman [Balzer99] and Wells and Pazandak [Wells01]). There are many technologies available for monitoring systems, and we build on those. However, they do not by themselves solve the hard problem of mapping from code to more abstract models. In previous work, we developed an infra-structure doing certain kinds of abstraction [Garlan03]. However, this approach was limited to observing properties of a system and reflecting them in a pre-constructed architectural model. In this work, we show how to create that model.

The work by Dias and Richardson [Dias03] uses an Extensible Markup Language (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 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 [Kaiser03]. 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 [Reiss03, Walker98, Walker00, and Zeller01]. 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 [Ernst01]. This is somewhat different from detecting architectural structure.

Madhav [Madhav96] describes a system that allows Ada 95 programs to be monitored dynamically to check conformance to a Rapide architectural specification [Luckham96]. 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 the approach used by Aldrich and colleagues [Aldrich02].

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 [Allen94] and Vieira and colleagues [Vieira00]). These approaches, however, do not link architecture to an executing system.

3 DiscoTect

3.1 Technical Challenges

Any approach that supports dynamic discovery of architectures must address three challenges:

(1) Monitoring: observing a system's runtime behavior,

(2) Mapping: interpreting that runtime behavior in terms of architecturally meaningful events, and

(3) Architecture Building: 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 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 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 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, there is no single architectural style or pattern that can be used for all systems. For example, the use of 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.

3.2 The DiscoTect Approach

To address these concerns, we have adopted the approach illustrated in Figure 1. Events captured from a running system are first filtered to select the subset of system observations that must be considered. The resulting stream of events is then fed to the DiscoTect Engine. The DiscoTect Engine takes in a specification of the mapping, written in a language called DiscoSTEP (Discovering Structure Through Event Processing). The DiscoTect engine constructs a Colored Petri Net from the mapping 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 an architectural model, which can then be displayed to a user or processed by architecture analysis tools. We now elaborate each of the three main components in turn:

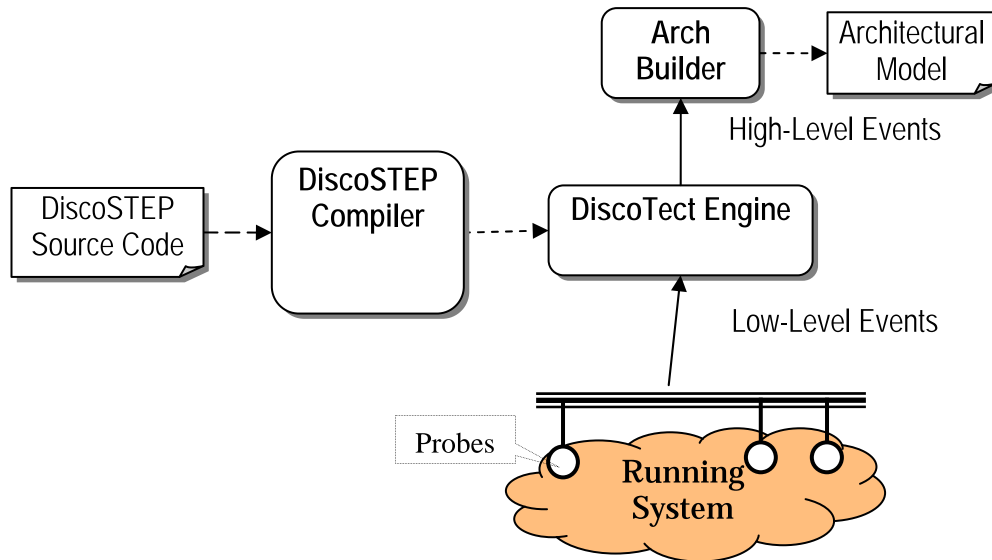

Figure 1. The DiscoTect Architecture

1. **DiscoSTEP Mapping Specification.** We have developed a language, called DiscoSTEP that allows the specification of mappings between low-level and architecture events. The execution of these mappings is defined using Colored Petri Nets – we provide a translation from DiscoSTEP mappings to Colored Petri Nets. This translation is defined in 3.5.

2. **DiscoTect Runtime Engine.** The run-time engine takes, as inputs, events from the running program and a DiscoSTEP specification and runs the DiscoSTEP specification to produce architecture events. System runtime events are first intercepted and converted into XML (Extensible Markup Language) [XML] streams by *Probes*. The resulting stream of events is then fed to the DiscoTect Runtime Engine which uses the DiscoSTEP specification to recognize interleaved patterns of runtime events and, when appropriate, outputs a set of architectural events.

3. **Architecture Builder.** The architecture builder takes architectural events and incrementally constructs an architectural description, which can then be displayed to a user or processed by other architecture analysis tools.

3.3 DiscoSTEP Language Requirements

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 refer to as an *architectural style* [Allen96]), we create a description in a language called DiscoSTEP. This description 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 must have its own mapping. However, a significant benefit of our approach is that these mappings can be reused across programs that are implemented in the same style.

To dynamically discover architectures from running systems, DiscoTect needs to be instructed how to interpret system-level runtime events so that it can generate corresponding high-level events and eventually to reconstruct the architectures. Such instructions are given by event mapping specifications written in our DiscoSTEP language. The event processing language needs to address a number of concerns.

First, mappings between low-level system observations and architectural events are not usually one-to-one. Many low-level events may be completely irrelevant to the architecture discovery process. 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 some 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 that a language providing a simple one-to-one (event to event) mapping is not sufficient. Therefore, the language needs *to allow M-N mappings between events*, and needs to *be able to keep track of information from one event so that the information can be used in subsequent stages to recognize architectural events*.

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, given a set of implementation conventions and a vocabulary of architectural element types and operations, we need to be able to *provide a mapping specification that captures the way in which runtime events following the implementation style should be interpreted as operations on elements of the architectural style*. In this way, we can build a library containing mappings between implementation style and architectural style pairs. A significant benefit for this is that these mappings can be reused across programs that are implemented in the same style.

Fourth, complex systems often involve more than a single implementation style and architectural style. This requires the event processing language *to provide an easy way to assemble code units each of which handles the mapping for a specific implementation and architectural style pair*.

3.4 Informal Introduction to DiscoSTEP

DiscoSTEP specifies how to map the system-level events to architectural-level events. To discover the architecture of a system, a program written in the DiscoSTEP language is compiled into byte code and fed to the DiscoTect engine. The DiscoTect engine processes the runtime events generated by the running system and generates architectural events on the fly. The architectural events can be further consumed by an architectural builder to construct the architecture.

A DiscoSTEP specification has three main ingredients:

1. **Events.** To allow DiscoTect to be flexible with respect to the types of events it can consume and produce, a DiscoSTEP specification imports XML Schema definitions of events.
2. **Rules.** Rules specify how to map a series of system events into architectural events.
3. **Compositions.** To complete a DiscoSTEP specification, compositions of rules are defined that allow complex sequences of events to be constructed.

In this section, we informally describe these three components. We then use a simple example to illustrate how they form a language to instruct event handling.

3.4.1 Events

3.4.1.1 Representation

An event is a message that indicates something has happened. A running software system involves many kinds of events including method calls, CPU utilization, network

bandwidth consumption, memory usage, and etc. We call those events runtime events. Our system deploys *probes* to collect runtime events. A probe can be a program that monitors the running system's resource consumption, such like network, memory and CPU utilization. It can also be code fragments injected into the target system using certain tracing technique, for example AspectJ [Kiczales01], AspectC++ [Spinczyk02].

In addition to runtime events that provide us the forensics evidence for tracing and profiling the running system, we define another type of events called architectural events. Architectural events are generated by our discovery system based on reasoning about runtime events. They are then consumed by an architecture builder to construct the system architecture. In some sense, architecture discovery is a data mining process in which a large number of runtime events are processed to generate architectural events that outline the system.

Both runtime events and architectural events need to be represented in machine-readable form. Instead of forcing event representation into a fixed format, we adopt XML to specify events and use XML schema as the type system to check if an event has a valid form/type. For example, the following XML schema defines events that indicate method calls:

```
<element name="call">
   <complexType>
      <attribute name="method_name" type="string" />
      <attribute name="callee_id" type="string" />
      <attribute name="return_id" type="string" />
   </complexType>
</element>
```

Figure 2. Defining Event Schema for use in DiscoSTEP.

This XML schema specifies the format of a “call” event: A well-formed “call” event should have the name “call” and can have all or part of the following attributes: the name of the method, the object id (an unique identification assigned to an instance at runtime) of the method callee, and the object id of the method return. Below is an example of a “call” event:

```
<call method_name="java.net.ServerSocket.accept"
      callee_id="19efb05"
      return_id="1d1acd3" />
```

Figure 3. An example “call” event.

By adopting XML we get the flexibility of customizing events, and by employing XML schema as the type system, we obtain the capability of type checking events to make sure they meet constraints specified by schema.

We reiterate that the structure of these events is not assumed by DiscoSTEP, but rather are specified along with the DiscoSTEP mapping. Therefore, if it is important to the mapping to know the calling object, this can be specified in the event and manipulated by the DiscoSTEP specification. (Of course, the monitoring infrastructure used would need to be able to provide enough information to form this event.)

3.4.1.2 Declaration and Import

From the language’s perspective, an event is a value of certain type and the type corresponds to an element of an XML schema, such as “call” in the above example. We declare events by binding an event variable with an event type (schema element). But before that, we need to import the schema from a schema file. Suppose we define the schema element “call” in a schema file “sys_event.xsd”, the following code segment in

our event processing language imports the schema and declare the variable $e as a “call” event:

```
import <sys_event.xsd>
…
call $e;
```

Figure 4. Importing Event Specifications in DiscoSTEP.

3.4.1.3 Input and Output

Typically our event processing engine takes runtime events as input events and produce architectural events as output events. For clarity, and for the purposes of checking the correctness of a DiscoSTEP specification, input and output event types are explicitly defined before they can be used to declare event variables. For example, the following code block defines “call” as an input event type and defines “create_component” as an output event type:

```
input {
    call;
    …
}
output {
    create_component;
    …
}
```

Figure 5. Specifying Input and Output Events to a DiscoSTEP rule.

3.4.2 Rules and Compositions

Rules are event handlers that decide what events should be processed and what the handling strategies are. Rules are composed of *inputs*, *outputs*, *triggers* and *actions*. Input and output blocks declare input events that this rule cares about and the output events that this rule can possibly generate. Triggers are predicates over the input events; actions are assignments to the output events. When a trigger returns true upon the arrival of input

events, the actions in the corresponding rule are activated to instantiate the output events. Since events are represented in XML, we borrow a well-defined XML Query language called XQuery [XQUERY] to describe triggers and actions.

Multiple rules can be assembled into a *composition* to handle event sequences. In a composition, rules are connected to each other by a set of input/output bindings. DiscoTect provides two forms of binding, directional and bidirectional (denoted as -> and <-> in the concrete syntax). The purpose of a bidirectional binding is to allow rules to fire on inputs without consuming the event (the event is implicitly reproduced as an output). This allows, for example, a rule for recognizing ports to generate multiple ports for the same component. Bidirectional bindings are shorthand for two directional bindings.

Section 3.4.4 provides concrete examples of rules and compositions.

3.4.3 Informal Runtime Semantics

Informally, DiscoTect runs DiscoSTEP specifications in the following sequence:

1. If an event is received by DiscoTect, associate those events with any rules that may accept that type of event.
2. For any rule that has a value for each of its inputs, evaluate the trigger.
3. If a trigger matches for a set of input events, execute the action with that set of events.
4. For output events that are composed with other rules, send the event as an input to that rule. For output events that are not composed with other rules, emit them from DiscoTect.

For each rule, an input event can be considered to be a set of events that match that type. Triggers match up events to be used in the rules. Formally, we model events as colored

tokens (where the color is given by the type), and each rule as a transition. For each event, a token is generated and put in the corresponding place before a transition. Rules consume tokens and put them in places after transitions. We discuss this formal definition in Section 3.5.

3.4.4 An Example DiscoSTEP Specification

To illustrate the concept of events and the use of rules and compositions, we now profile a simple program written in Java. In doing so we illustrate how to specify events using DiscoSTEP, and how DiscoTect uses this specification to generate an architectural description. The example is a simple system that implements a chat server. The chat server creates a server socket and announces its intention to accept connections. When a client connects to the waiting server, a new thread (of the type ClientThread) is started, which forwards all messages from that client to all connected clients. The source code for this application is presented in Figure 15 of the Appendix.

While this system is extremely simple, it allows us to illustrate the concepts of DiscoSTEP. In later sections we will describe more complex case studies. In the remainder of this section, we will specify how we instrumented this system, describe its DiscoSTEP specification, and discuss how DiscoTect processes this specification to produce an architectural description.

The architectural description for this system is based on a simple client-server style. Informally, it is constructed as follows: when the server is created in the program, this maps to a server type component in the architecture; ClientThreads map to client components, and the socket connection maps to an architectural connector between each client and server.

3.4.4.1 Instrumentation

The act of instrumenting a system to produce runtime events is not a novel aspect of DiscoTect. In fact, where possible, we use off-the-shelf technologies to instrument the system. In the Java-based systems that we have studied, we have used AspectJ to define instrumentation *aspects* that are weaved into the compiled bytecode of the Java programs. These aspects emit events when methods of interest are entered or exited, and when objects are constructed.

The aspects can reflectively retrieve information about the runtime environment of, for example, a call, to ascertain the calling object, the instance of the object that was called, the argument values and types that were passed to the method, the method signature, etc. The aspects are written to emit XML elements that conform to a schema expected by DiscoTect. For example, to instrument the ChatServer, we weaved in aspects to emit events when methods were called and when objects were constructed.

3.4.4.2 Runtime Events

Two types of runtime events were collected from this running system: call events and init events. A call event is reported when a method is invoked. Similarly, an object instantiation produces an init event. Take the following two events for example:

```
<init constructor_name="ServerSocket" instance_id="10">

<call method_name="ServerSocket.accept"
      callee_id="10" return_id="11" />
```

An init event is generated when

```
ServerSocket ss = new ServerSocket(1111)
```

is executed; a call event is triggered by an execution of a method call. For example, the call event above is emitted by the following statement execution.

```
Socket socket = ss.accept()
```

Because multiple ClientThreads can run concurrently, some of the runtime events, such as InputStream.read and OutputStream.write, show up in random order and hence may be interleaved with each other.

A fuller trace of the events that we retrieved when running the program is available in Appendix A. These events can be fed into DiscoTect either in real time or off-line, after the program has completed running.

3.4.4.3 DiscoSTEP Program

A DiscoSTEP program that specifies how to handle the interleaved events between the client and the server was specified to formally capture how to map system events into architectural events. The full DiscoSTEP program for this example is given in Appendix B; in this section we discuss some of the rules and how these are combined with the event trace to produce the architecture. DiscoTect takes the runtime events from the ChatServer to produce architectural events that construct a Client Server style representation of the system.

Figure 6 shows a fragment of a DiscoSTEP program that includes two rules, and how they are composed. The CreateServer rule constructs and architecture Server component. It takes the input event under inspection to be an init event named $e. The output events include the string event $server_id and the create_component event $create_server. The condition for triggering this rule is that the constructor_name attribute of $e contains the string "ServerSocket". If the rule is triggered, the following

action is taken: $server_id is assigned the id of the object constructed in the init event, and an architecture event that constructs a server component named with the id of the newly created instances is assigned to $create_server.

```
Figure 6rule CreateServer {
    input { init $e; }
    output { string $server_id; create_component $create_server; }
    trigger {? contains($e/@constructor_name, "ServerSocket") ?}
    action = {?
        let $server_id := $e/@instance_id;
        let $create_server :=
            <create_component name="{$server_id}"
                            type="ServerT" />;
    }
}
rule ConnectClient {
    input { call $e; string $server_id; }
    output {
        create_component $create_client;
        create_connector $create_cs_connection;
        string $client_id;
    }
    trigger {?
        contains($e/@method_name, "ServerSocket.accept")
         and $e/@callee_id = $server_id
    ?}
    action = {?
        let $client_id := $e/@return_id;
        let $create_client :=
            <create_client name="{$client_id}" type="ClientT" />;
        let $create_cs_connection :=
            <create_connector name=concat($client_id,"-",$server_id)
                type="CSConnectorT"
                end1="{$server_id}" end2="{$client_id}" />;
    ?}
}
composition {
    CreateServer.$server_id -> ConnectClient.$server_id;
    …
}
```

Figure 6. The DiscoSTEP rule to create a Server component.

The `$server_id` output from the `CreateServer` rule is fed to the `ConnectClient` rule, which has two *inputs*: `$e` and `$server_id`. Once the these inputs are received by `ConnectClient`, the trigger will check to see if any call events are calls to `ServerSocket.accept`. If so, output events `$client_id`, `$create_client` and `$create_cs_connection` are assigned appropriate values to construct both client component and the connector connecting it with the previously created server component. Instead of being specific to this particular ChatServer program, our client server event processing program is generic enough to be applicable to any client server applications implemented with the same style (with, at the most, some minor changes in the triggers). Both the compositions and the rules are well encapsulated. Rules are self-contained specifications, communicating with each other via inputs and outputs; compositions function as glue that assemble the rules. We can reuse compositions by applying them to a different system, and reuse rules by assembling them with a different composition (and adding new rules if necessary).

3.4.5 Satisfying the Requirements

In this section we revisit the requirements for DiscoSTEP that we introduced in Section 3.3 and discuss how DiscoSTEP meets these requirements.

- *Allow M-N mappings between events.* Since a DiscoSTEP rule can have an arbitrary number of inputs and outputs, this requirement is simply met by DiscoSTEP.
- *Keep track of information for use in subsequent stages.* Input events and output events are essentially data structures that can be passed from one rule to the next.

These data structures are used to store and accumulate information that can be passed between rules. Compositions define how this state is passed between rules.

- *Cope with concurrent states.* The informal execution semantics defined in Section 3.4.3 describe how input events are propagated to each rule that can accept an event of that type. In this way, these events can start multiple execution threads for rules to cope with concurrent states. A rule will wait until it gets a set of input events that match a trigger before firing. In this way, interleaved threads of "conversation" can be disentangled.
- *Allow the assembly of code unit.* Though not described in detail in this paper, the abstract syntax of DiscoSTEP specifies that a composition itself may have input and output events, as well as subcompositions. In this way, compositions can be combined hierarchically to form more complex mappings. So, for example, it is possible to take a composition that identifies mapping between the usage of files in a system to a data repository architectural style, and combine that with a mapping that recognizes the construction of a pipe-filter architecture to define the mapping for a pipe-filter system that retrieves and stores data in files.

In toto, meeting the above requirements means that DiscoSTEP meets the final requirement: *to provide a mapping specification that captures the way in which runtime events following the implementation style should be interpreted as operations on elements of the architectural style*. This requirement is met by specifying a DiscoSTEP mapping.

3.5 Formal Definition of DiscoSTEP

To define the execution semantics of a DiscoSTEP program, and to formally explain how the mappings are interpreted, we use Colored Petri Nets [Jenson94]. In [Yan04] we informally described the semantics of DiscoTect mappings in terms of state machines. However, this semantics was awkward because of the need to retain multiple active states in the state machine in order to model the concurrency in the model. We believe that Colored Petri Nets are the most appropriate formalism for describing the semantics of DiscoSTEP mappings because their tokens provide a rich way of representing the concurrent states of the system.

In this section we formally describe the DiscoSTEP language. We begin by describing an abstract syntax of DiscoSTEP, which is suitable for formal specifications and proofs, followed by typechecking rules that ensure a DiscoSTEP program is meaningful. We then describe DiscoSTEP's semantics through rewriting rules that transform a DiscoSTEP program into a Colored Petri Net.

3.5.1 DiscoSTEP Abstract Syntax

The concrete syntax for DiscoSTEP, which we have been using up to this point, is given in Figure 17 of Appendix A. Although this syntax is easily readable, its lack of structure makes it poorly suited for formal analysis, including rules for defining DiscoSTEP's type system and semantics. Therefore, we describe an Abstract Syntax for DisocSTEP that is more amenable to formal specifications.

Conceptually, a DiscoSTEP program is a 3-tuple (T_{in}, T_{out}, C_{main}). Here, T_{in} and T_{out} represent the sets of input and output events declared in the input and output clauses of a DiscoSTEP program. Without loss of generality, we assume that a DiscoSTEP program

is made up of one top-level component C_{main}. We further decompose component declarations C into rules, as follows:

- A composition C is a tuple:

  $$C = (c, \overline{R}, \overline{C'}, \overline{(v_o, v_i)})$$

  where c is a name uniquely identifying the composition in the program. We represent a sequence with an overbar, so that $\overline{R} = R_1 \ldots R_n$ is the set of rules defining the behavior of C; $\overline{C'}$ is the set of sub-compositions of C; and $\overline{(v_o, v_i)}$ is a set of connections, each of which connects an output variable v_o of some rule $R_j \in \overline{R}$ and some input variable v_i of some rule $R_k \in \overline{R}$.

- A rule R is a tuple:

  $$R = (r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, pred(\overline{v_{in}}), \overline{(v_{out}, exp(\overline{v_{in}}))})$$

  where r is a name uniquely identifying the rule in the program; $\overline{v_{in}}$ and $\overline{v_{out}}$ are input and output variables of the rule; $\overline{t_{in}} \in T_{in}$ and $\overline{t_{out}} \in T_{out}$ are the type of the input and output variables $\overline{v_{in}}$ and $\overline{v_{out}}$, respectively; $pred(\overline{v_{in}})$ is an XQuery predicate that may only use variables from the set of input variables, and $\overline{(v_{out}, exp(\overline{v_{in}}))}$ is an assignment of XQuery expressions over the set of input variables $\overline{v_{in}}$ to the output variables $\overline{v_{out}}$.

We do not directly model the semantics of XQuery, as they are defined elsewhere [W3C04]. We also assume that all variable and rule names are globally unique.

3.5.2 Type Checking

Not every DiscoSTEP program allowed by the syntax in the previous section makes sense. For example, one could write a composition that connects an output of a certain type to an input of a different type without breaking the syntax. We use a set of typechecking rules to ensure that a DiscoSTEP program is *well-typed*. A well-typed DiscoSTEP program has sensible runtime behavior.

$$\frac{v_1{:}T \in \Gamma \quad v_2{:}T \in \Gamma}{\Gamma \vdash (v_1, v_2)\ \text{ok}}\ \text{T-CONN}$$

$$\frac{\Gamma = \overline{v_{in}} : \overline{t_{in}}, \overline{v_{out}} : \overline{t_{out}} \qquad \Gamma \vdash \text{pred}(\overline{v_{in}}) : \text{bool} \qquad \Gamma \vdash \overline{\text{exp}(\overline{v_{in}})} : \overline{t_{out}}}{\Gamma \vdash (r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \text{exp}(\overline{v_{in}}))}) : \overline{T_{in}} \times \overline{T_{out}}}\ \text{T-RULE}$$

$$\frac{\overline{\Gamma_R} \vdash \overline{R}\ \text{ok} \qquad \overline{\Gamma_C} \vdash \overline{C'}\ \text{ok} \qquad \overline{\Gamma_R}, \overline{\Gamma_C} \vdash \overline{(v_1, v_2)}\ \text{ok}}{\overline{\Gamma_R}, \overline{\Gamma_C} \vdash (c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)})\ \text{ok}}\ \text{T-COMP}$$

Figure 7. The full set of type inference rules for DiscoSTEP.

Figure 7 shows the typechecking rules for DiscoSTEP, presented in a form that is standard in the programming language literature. Most of the rules have one or more *premises*, written above the line; if all of these are valid, then we can conclude that the *conclusion*, written below the line, holds.

The premises and conclusions are *judgments* of the form $\Gamma \vdash C\ \text{ok}$ stating that a composition C is well-formed given a list Γ mapping variables in scope to their types (and similar for rules R and connections (v_1, v_2)).

The first rule states that a connection between variables v_1 and v_2 is ok if the typing assumptions Γ tell us that they have the same type T. Thus this rule would prohibit ill-formed connections as described above.

A **CP-net** is a tuple CPN = (Σ, P, T, A, N, Col, G, E, I) where:
(i) Σ is a finite set of non-empty **types**, also called color sets.
(ii) P is a finite set of **places**.
(iii) T is a finite set of **transitions**.
(iv) A is a finite set of **arcs** such that:

- $P \cap T = P \cap A = T \cap A = \emptyset$.

(v) N is a **node** function. It is defined from A into $P \times T \cup T \times P$.
(vi) Col is a **color** function. It is defined from P into Σ.
(vii) G is a **guard** function. It is defined from T into expressions such that:

- $\forall t \in T: [Type(G(t)) = B \wedge Type(Var(G(t))) \subseteq \Sigma]$.

(viii) E is an **arc expression** function. It is defined from A into expressions such that:

- $\forall a \in A: [Type(E(a)) = Col(p) \wedge Type(Var(E(a))) \subseteq \Sigma]$

where p is the place of N(a).
(ix) I is an **initialization** function. It is defined from P into closed expressions such that:

- $\forall p \in P: [Type(I(p)) = Col(p)]$.

Figure 8. The definition of Colored Petri Nets, from [Jensen94].

The second rule states that a rule R is ok if we compute a set of typing assumptions Γ from the types of the input and output variables, and if using those assumptions we can use XQuery's type system to conclude that the predicate expression has a boolean type and that the output expression for each output variable v_{out} has the type t_{out} of that variable. We do not model XQuery's type system directly, as this is defined elsewhere, but we assume the presence of a judgment form $\Gamma \vdash e : T$ stating that XQuery expression e has type T given assumptions Γ [W3C04].

The final rule states that a composition is ok if all of its constituent rules, sub-compositions, and connections are ok. The connections are typechecked using the combined typing assumptions of all the constituent rules and sub-compositions, since in fact the connections might reference any variables in those parts.

Submitted for publication.

3.5.3 Translational Semantics of the DiscoSTEP

According to [Jensen 94], a CP-net has the definition presented in Figure 8. The translation semantics of the DiscoSTEP language define how to convert a DiscoSTEP program into a CP-net.

Figure 9 gives the full set of translational semantics for mapping between DiscoSTEP and a CP-net, given as a set of functions from a piece of DiscoSTEP syntax to one of the elements of the CP-net. The rules may be applied recursively to form the corresponding sets in the CP-net definition. For example, the first rule in

Figure 9 gives instructions on how to form the set T of types in a CP-net. If the function is applied to a DiscoSTEP rule, then it is the union of all types used in the rule. If it is applied to a composition, then it returns the union of the sets of types that result from applying the function recursively to all the rules and sub-compositions defined in the composition. Thus, for rule CreateServer in Figure 6 the function GetType(CreateServer) returns the colors init, create_server, and string.

Two pieces of notation are used in the rules. First, the notation $[\overline{v_1} \doteq \overline{v_2}]\ e$ means, for each pair (v_1, v_2), choose one as a canonical representative for the pair and replace the other with the canonical representative in e. This unification is used in the getPlace rule and others to ensure that only one place is created for each connected pair of variables in the source text. Second, to construct the names of arcs in the CP-net, we concatenate the name of a rule and a variable together with the :: operator, as in v_{in}::r.

$$\text{fun GetType}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = \overline{t_{in}} \cup \overline{t_{out}}$$
$$\mid \text{GetType}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{\text{GetType}(R)} \cup \overline{\text{GetType}(C')}$$

$$\text{fun GetTransition}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = r$$
$$\mid \text{GetTranstion}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{\text{GetTransition}(R)} \cup \overline{\text{GetTransition}(C')}$$

$$\text{fun GetPlace}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = \overline{v_{in}} \cup \overline{v_{out}}$$
$$\mid \text{GetPlace}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetPlace}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetPlace}(C')}$$

$$\text{fun GetArc}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = \overline{v_{in} :: r} \cup \overline{r :: v_{out}}$$
$$\mid \text{GetArc}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetArc}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetArc}(C')}$$

$$\text{fun GetNode}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) =$$
$$\overline{(v_{in} :: r, (v_{in}, r))} \cup \overline{(r :: v_{out}, (r, v_{out}))}$$
$$\mid \text{GetNode}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetNode}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetNode}(C')}$$

$$\text{fun GetColor}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = \overline{(v_{in}, t)} \cup \overline{(v_{out}, t)}$$
$$\mid \text{GetColor}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetColor}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetColor}(C')}$$

$$\text{fun GetGuard}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = (r, \text{pred}(\overline{v_{in}}))$$
$$\mid \text{GetGuard}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetGuard}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetGuard}(C')}$$

$$\text{fun GetAction}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = (\overline{r :: v_{out}}, \overline{\exp(\overline{v_{in}})})$$
$$\mid \text{GetAction}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetAction}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetAction}(C')}$$

$$\text{fun GetInit}(r, \overline{(v_{in}, t_{in})}, \overline{(v_{out}, t_{out})}, \text{pred}(\overline{v_{in}}), \overline{(v_{out}, \exp(\overline{v_{in}}))}) = \overline{(v_{in}, \emptyset)} \cup \overline{(v_{out}, \emptyset)}$$
$$\mid \text{GetInit}(c, \overline{R}, \overline{C'}, \overline{(v_1, v_2)}) = \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetInit}(R)} \cup \overline{[\overline{v_1} \doteq \overline{v_2}]\text{GetInit}(C')}$$

Figure 9. The Translational Functions for Mapping between DiscoSTEP and CP-net.

Taken together, the CP-net for a given composition $C = (c, \overline{R}, \overline{C'}, \overline{(v_{in}, v_{out})})$ is formed using the following translation rule:

$$\frac{\begin{array}{l}\Sigma = \text{GetType}(C),\quad P = \text{GetPlace}(C),\quad T = \text{GetTransition}(C) \\ A = \text{GetArc}(C),\quad N = \text{GetNode}(C),\quad Col = \text{GetColor}(C) \\ G = \text{GetGuard}(C),\quad E = \text{GetAction}(C),\quad I = \text{GetInit}(C)\end{array}}{C \mapsto (\Sigma, P, T, A, N, Col, G, E, I)}$$

translation is done.

3.5.4 Formally Modeling the Example

The ChatServer DiscoSTEP program uses the following types: string, init, call, create_component, create_connector, and update_component. By applying GetType, we obtain can derive the color sets for the CP-net as:

Σ = {string, init, call, create_component,
create_connector, update_component }.

The next step is to obtain the set of transitions for the CP-net. By applying GetTransition, the six rules are translated into six corresponding CP-net transitions:

T = {CreateServer, ConnectClient, ClientIO,
ClientRead, ClientWrite, UpdateServer}

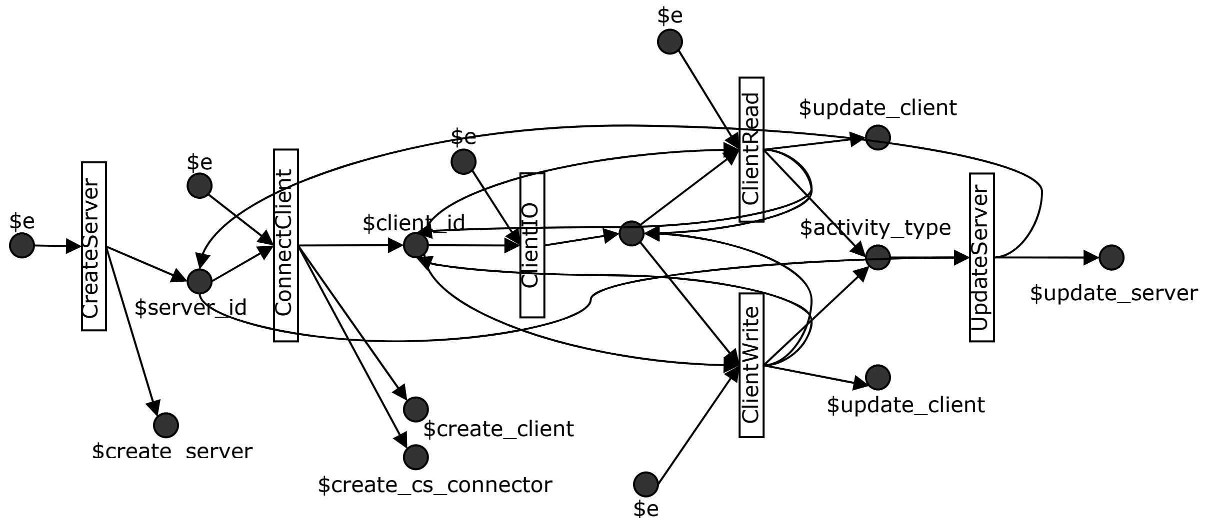

Figure 10. CP-net Places, Arcs and Node functions translated from inputs and outputs.

By applying GetPlace, GetArc, and GetNode, the inputs and outputs are translated into CP-net places, arcs and node functions. Using GetGuard and GetAction, the triggers and actions are translated into CP-net guards defined from transitions into predicates, and arc expressions defined from arcs to *XQuery* expressions.

Figure 10 shows the resulting net. Note that the backward arcs from, for example, the ClientRead transition to the $client_id place are formed through the unification process described above, because the $client_id output of ConnectClient is bound to the inputs of more than one rule (ClientIO, ClientRead, and ClientWrite).

4 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: We use various existing probing technologies to extract monitoring events. In this section, we will illustrate the use of AspectJ [Kiczales01], 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 strings which are placed on a JMS event bus to be consumed by DiscoTect.

Mapping: The implementation of the DiscoTect Engine follows the design in Section 4. During initialization, the Engine parses the DiscoSTEP definition and activates the transitions. Then it keeps scanning the event stream sent from the probes, producing colored tokens for each event. A token is placed in the corresponding place that can accept that color. Once there is a token at each of the input places of a transition, the guard for that transition is evaluated. If a guard condition is satisfied, the actions for that transition are evaluated and the corresponding tokens placed on the output places of the transition.

Architecture Building: We represent architectures using the Acme architecture description language [Garlan00] (although we are not restricted to this language; in

principle any architecture description language could serve in this capacity). Operations on Acme architectures are defined in a library that provides operations that form building blocks of architectural actions. To connect to our existing architectural tools, DiscoTect produces architectural events formatted as XML strings that are forwarded by the AcmeStudio Remote Control plugin, communicating over Java RMI, to incrementally construct the architecture. AcmeStudio [Schmerl04] is an architecture development environment that is primarily used for constructing architectures at design time. 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.

5 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 [Kazman03]. 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 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 11 shows the (informal) runtime architecture of AAMS as presented in [Kazman 03]; the following description of the runtime is also based on the description in this paper. The Simulation Controller forms a simulation from a description of 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

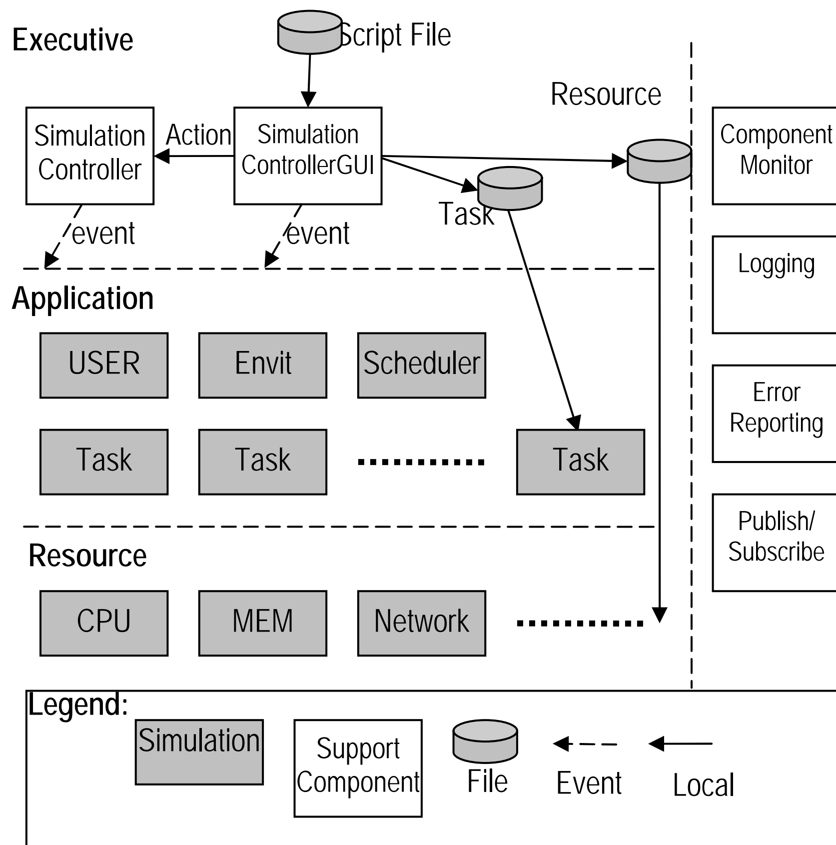

Figure 11. Documented runtime view of AAMS

I/O, Error Reporting and Logging are available via publish/subscribe to any simulated object.

5.1 Design of AAMS DiscoSTEP program

In this section we present the steps taken to produce the DiscoSTEP program to discover the AAMS architecture model. Typically, writing these programs is a process of starting with fairly generic rules to discover components and connections, and then refining these rules 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 [Kazman03].

To develop the final DiscoSTEP program, we first produced rules that merely observed object creation and interaction (through procedure calls). We then refined this set of rules to classify objects into their architectural counterparts (e.g., Resource, Task, etc.).

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.

5.2 The Discovered Architecture

Applying the above state machine to a running instance of AAMS yields the architectural model in Figure 12. (We have laid out this model to enable easier comparison with the view in Figure 11.) By comparison with Figure 11, 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 12 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

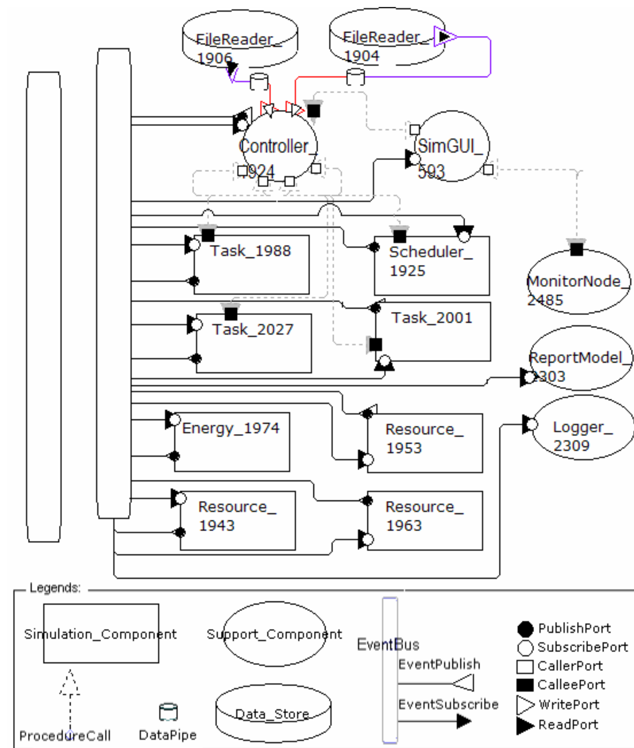

Figure 12. The Discovered Architecture of AAMS.

11), the controller reads two files. This discrepancy could cause errors during evolution if the AAMS system was to work in a distributed environment. The evolution might require that the file reading components run on the same computer as that containing the files. The documented architecture would suggest that Simulator GUI is the component that should stay with the files, when in fact it is the Controller component according to the implementation.

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

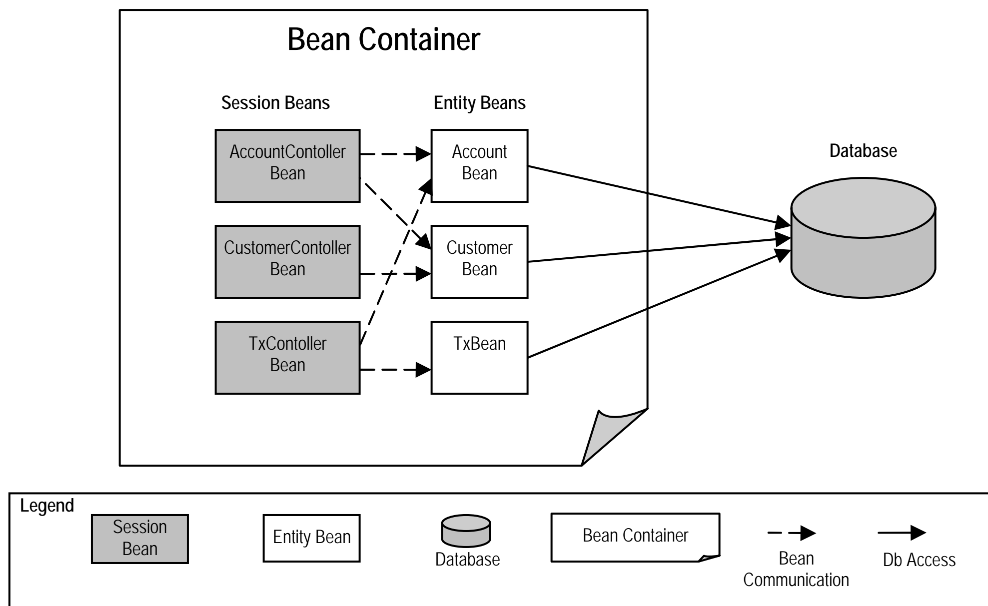

Figure 13. Documented architectural view of Duke's Bank Application

and Environment components are due to the fact that these represent user interaction, and are not actual components in the implementation.

6 EJB Case Study

In this section we present a second case study to determine the run-time architecture of the Duke's Bank Application – a simple EJB (Enterprise JavaBeans) 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 [J2EE], which enables us to compare the actual discovered architecture with the one presented in the documentation.

We wrote an aspect which 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.

Figure 13 gives a high-level view of how the components interact in the Duke's Bank system as presented in [J2EE]. The EJB application has three session beans: AccountControllerBean, CustomerControllerBean, and 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. CustomerControllerBean 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 accesses TxBean to keep records of the transactions.

6.1 Design of the EJB State Machine

In this section we present the steps taken to produce the DiscoTect state machine to discover 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, database etc. These component types are based on the EJB specification found in [EJB].

As we did in the previous case study, we first produced primitive rules that merely observed object interaction and creation (through procedure calls and object instantiations). We then refined these rules to classify objects into their architectural counterparts (e.g., Beans, Bean Containers, Database etc.) 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 in the following way: 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 JDBC (Java Database Connectivity) [JDBC], we monitored the standard JDBC APIs to uncover the connections between the beans and the database; the interactions between the beans were also monitored and represented as connectors linking them together.

6.2 The Discovered Architecture

Applying the state machine just described to a running instance of Duke's Bank yields the architectural model in Figure 10. We have organized the layout this model for better comprehensibility. We can make the following observations based on this process.

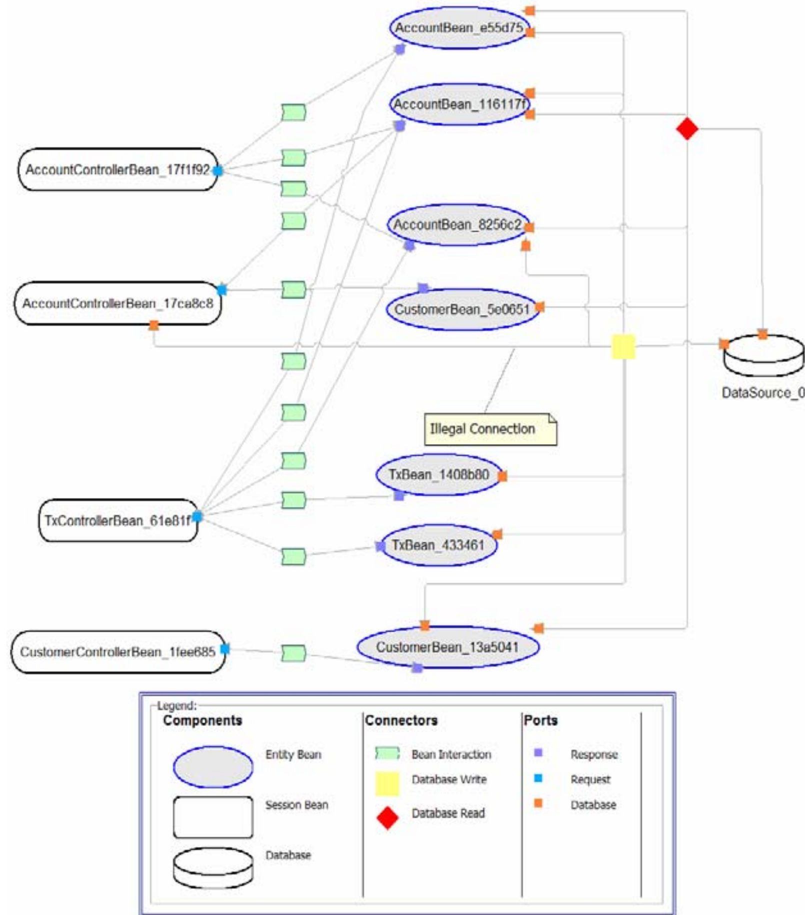

Figure 14. Discovered architecture of Duke's Bank

1. *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.

2. *Verification of Bean Interplay*. The interactions between the beans shown in Figure 14 are consistent with those described in the architecture shown in Figure 13: there are communication channels between AccountControllerBean and AccountBean, AccountControllerBean and CustomerBean, CustomerControllerBean and

CustomerBean, TxControllerBean and TxBean, TxControllerBean and AccountBean.

3. *Discrepancies in Database Access*. Figure 13 does not show any connections between the session beans and the database, which implies that all database access goes through the entity beans. This is consistent with Sun's EJB specification [EJB]. However a "database write" connector appeared in the discovered architecture. Further source code analysis (performed manually) confirmed that AccountControllerBean does directly write to the database. As discussed in the previous section, identifying communication "backdoor" connections like this is useful for architectural analysis and to ensure architectural conformance.

7 Conclusions and Future Work

In this paper we described an approach to "discovering" the architecture of a running system that uses 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 a view in that architectural style. The mapping itself defines a novel form of behavior specification (realized as a Colored Petri Net) that relates low-level events to architecturally-significant actions. The power of Petri Nets is used to model the current threads of event recognition, allowing us to disentangle the interleaved sequences of low-level events that contribute to higher-level architectural behavior.

There are a number of advantages of this approach. 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 run-time monitoring. 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. For example, although not described in this paper, we have been able to successfully discover the Pipe/Filter architecture of a system implemented using different pipe conventions.

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. Fourth, the DiscoSTEP mapping needs to be created via an iterate-and-test paradigm, and hence the results are somewhat dependent on the skill of the creator of the recognizer. Thus our techniques are best viewed as one of several technologies that an architect must have in his arsenal of architecture conformance checking tools. For example, we believe that DiscoTect can be effectively combined with static analysis tools such as Dali [Kazman99] or Armin [O'Brien03] 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. As we have mentioned, we have experimented with a number of existing monitoring technologies for Java, and to some extent C++. However, monitoring technology for other kinds of implementations and system properties is an active research area that should continue to provide increasing capabilities in the future that we can leverage.

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. We can envision a large library of recognizers for common architectural frameworks, available, perhaps, as open source libraries, which would track the most common architectural frameworks in practical use.

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.

Fifth, while the approach we have outlined focuses primarily on recognizing architectural *structure*, we believe it could be easily extended to *architectural behavior.* For example, we can imagine using the same run-time abstraction techniques to check that the observed interaction between two components conforms to the protocol expected over the corresponding architectural connector. Similarly we might, observe timing behavior, which could be compared with an architectural specification of expected performance.

References

[Aldrich02] J. Aldrich, C. Chambers, and D. Notkin. "ArchJava: Connecting Software Architecture to Implementation," In Proceedings of the 24th International Conference on Software Engineering, 2002.

[Allen94] R. Allen, D. Garlan. Formalizing Architectural Connection. In Proceedings of ICSE 1994.

[Balzer99] R.M. Balzer and N.M Goldman. "Mediating Connectors," Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, 1999.

[Bass03] L. Bass, P. Clements, R. Kazman. Software Architecture in Practice, 2nd Edition, Addison-Wesley, 2003.

[Clements01] P. Clements, R. Kazman, M. Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, 2001.

[Clements02] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting Software Architectures: Views and Beyond, Addison Wesley, September 2002.

[Dias03] 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.

[EJB] Sun Microsystems. <http://java.sun.com/products/ejb/docs.html>

[Ernst01] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. "Dynamically discovering likely program invariants to support program evolution," IEEE Trans. on Software Engineering, 27(2), 2001.

[Garlan94] D. Garlan, R.J. Allen, and J. Ockerbloom. "Exploiting Style in Architectural Design," Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering (FSE 94), 1994.

[Garlan00] 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.

[Garlan02] 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.

[Garlan03] D. Garlan, S.-W. Cheng, B. Schmerl. “Increasing System Dependability through Architecture-based Self-repair”, in Architecting Dependable Systems, R. de Lemos, C. Gacek, A. Romanovsky (Eds). LNCS 2677, Springer-Verlag, 2003.

[J2EE] Sun Microsystems <http://java.sun.com/docs/books/j2eetutorial/index.html>

[Jackson99] D. Jackson and A. Waingold. “Lightweight extraction of object models from bytecode,” In Proceedings of the 1999 International Conference on Software Engineering, 1999.

[JDBC] Sun Microsystems <http://java.sun.com/products/jdbc>

[Jenson94] K. Jensen. “An Introduction to the Theoretical Aspects of Coloured Petri Nets.” In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, 230-272.

[Kaiser03] G. Kaiser, J. Parekh, P. Gross, and G. Veletto. “Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems,” Proceedings of 5th International Active Middleware Workshop, 2003.

[Kazman99] R. Kazman, and S.J. Carriere. “Playing Detective: Reconstructing Software Architecture from Available Evidence,” Journal of Automated Software Engineering 6(2), 1999.

[Kazman03] R. Kazman, J. Asundi, J.S. Kim, and B. Sethananda. “A Simulation Testbed for Mobile Adaptive Architectures,” Computer Standards and Interfaces, 2003.

[Kiczales01] G. Kiczales, E. Hilsdale, J. Huginin, M. Kersten, J. Palm, W. Griswold. “Getting Started with AspectJ,” In Communications of the ACM 4(10), October 2001.

[Luckham96] D.C. Luckham. “Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events,” Proceedings of the DIMACS Partial Order Methods Workshop, 1996.

[Madhav96] N Madhav. “Testing Ada 95 Programs for Conformance to Rapide Architectures,” Proceedings of Reliable Software Technologies – Ada Europe 96, 1996.

[Medvidovic00] Medvidovic N., and Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), pp. 70–93, 2000.

[Murphy95] G.C. Murphy, D. Notkin, and K.J. Sullivan. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," In Proceedings of 1995 ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995.

[O'Brien03] L. O'Brien, C. Stoermer, "Architecture Reconstruction Case Study," Software Engineering Institute Technical Note CMU/SEI-2003-TN-008, 2003.

[Perry92] D. Perry, A. Wolf. "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, 17(4), 1992.

[Reiss03] S. Reiss. "JIVE: Visualizing Java in Action (Demonstration Description)," Proceedings of 25th International Conference on Software Engineering, 2003.

[Schmerl04] B. Schmerl, and D. Garlan. "AcmeStudio: Supporting Style-Centered Architecture Development (Demonstration Description)," Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004.

[Shaw95] M. Shaw, R. Deline, D. Klein, T.L. Ross, D.M. Young, G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them." IEEE Transactions on Software Engineering 21(4), 1995.

[Shaw96] M. Shaw. and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

[Spinczyk02] O. Spinczyk, A. Gal, W. Schroder-Preikschat. "AspectC++: An Aspect-oriented Extension to the C++ Programming Language," Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile, and Embedded Applications, Volume 10, 2002.

[Taylor96] 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 Transactions on Software Engineering 22(6), 1996.

[Vestal96] S. Vestel. "MetaH Programmer's Manual, Version 1.09." Technical Report, Honeywell Technology Center, 1996.

[Vieira01] M. Vieira, M. Dias, D.J. Richardson. "Software Architecture based on Statechart Semantics," Proceedings of the 10th International Workshop on Component Based Software Engineering, 2001.

[Walker98] 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 Proceedings of OOPSLA'98,

[Walker00] 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 Proceedings of CASCON 2000.

[W3C04] W3C Consortium. “XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft 4 April 2005.” http://www.w3.org/TR/2005/WD-xquery-semantics-20050404/.

[Wells01] D. Wells and P. Pazandak. “Taming Cyber Incognito: Surveying Dynamic/Reconfigurable Software Landscapes,” Proceedings of 1st Working Conference on Complex and Dynamic Systems Architectures, 2001.

[XQuery] W3C <http://www.w3.org/TR/xquery/>

[XML] W3C <http://www.w3.org/XML/>

[Yan04] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. "DiscoTect: A System for Discovering Architectures from Running Systems," Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, May, 2004.

[Zeller01] A. Zeller. “Animating Data Structures in DDD,” In Proceedings of SIGCSE/SIGCUE Program Visualization Workshop, 2000.

Appendix A

```
Java code:

public class ChatServer {
  static class ClientThread extends Thread {
  private Socket socket;
  private Vector clients;
  public ClientThread(Socket socket,
                  Vector clients) {
    this.socket = socket;
    this.clients = clients;
    clients.addElement(socket);
  }
  public void run() {
    byte[] buf = new byte[1024];
    int len = 0;
    try {
      InputStream is = socket.getInputStream();
      while ((len = is.read(buf)) != -1) {
        // Broadcast the message to
        // all the clients
        for (int i = 0; i < clients.size(); i++) {
          OutputStream os =
           ((Socket) clients.get(i)).
             getOutputStream();
          os.write(buf, 0, len);
        }
      }
    } catch (IOException e) {
    } finally {
      clients.removeElement(socket);
      try {
        socket.close();
     } catch (IOException e) {}
    }
  }
  private static Vector clients = new Vector();
  public ChatServer() {
    ServerSocket ss = new ServerSocket(1111);
    while (true) {
      // Wait for clients to connect
      Socket socket = ss.accept();
      new ClientThread(socket, clients).start();
    }
  }
  public static void main(String[] args)
        throws IOException {
    new ChatServer();
  }
}
```

```
Runtime events:
<init constructor_name=”ServerSocket”
   instance_id=”10”>
<call method_name=”ServerSocket.accept”
   callee_id=”10” return_id=”11” …/>
…
<call method_name=”Socket.getInputStream”
   callee_id=”11” return_id=”1000” …/>
<call method_name=”ServerSocket.accept”
   callee_id=”10” return_id=”12” …/>
…
<call method_name=”InputStream.read”
   callee_id=”1000” …/>
<call method_name=”Socket.getOutputStream”
   callee_id=”11” return_id=”1001” …/>
<call method_name=”OutputStream.write”
   callee_id=”1001” …/>
<call method_name=”Socket.getInputStream”
   callee_id=”12” return_id=”1002” …/>
<call method_name=”InputStream.read”
   callee_id=”1002” …/>
<call method_name=”InputStream.read”
   callee_id=”1000” …/>
<call method_name=”Socket.getOutputStream”
   callee_id=”12” return_id=”1003” …/>
<call method_name=”OutputStream.write”
   callee_id=”1003” …/>
…
```

Figure 15. The Jave code for the ChatServer, and events produced through one run that are subsequently fed into DiscoTect.

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

```
  }
}
rule ConnectClient {
  input { call $e; string $server_id; }
  output {
    create_component $create_client;
    create_connector $create_cs_connection;
    string $client_id;
  }
  trigger {?
    contains($e/@method_name, "ServerSocket.accept") and $e/@callee_id = $server_id
  ?}
  action = {?
    let $client_id := $e/@return_id;
    let $create_client := <create_client name=$client_id type="ClientT" />;
    let $create_cs_connection :=
      <create_connector name=concat($client_id,"-",$server_id)
        type="CSConnectorT" end1=$server_id end2=$client_id />;
  ?}
}
rule ClientIO {
  input { call_event $e; string $client_id; }
  output { string $io_id; }
  trigger {?
    (contains($e/@method_name, "Socket.getInputStream") or
     contains($e/@method_name, "Socket.getOutputStream")) and
    $e/@callee_id = $client_id
  ?}
  action {? let $client_id := $e/@return_id; ?}
}

rule ClientRead {
  input { $e : call_event; $io_id : string; $client_id : string; }
  output { $update_client : update_component; $activity_type : string;}
  trigger {? (contains($e/@method_name, "InputStream.read") and $e/@callee_id = $io_id ?}
  action = {?
    let $update_client :=
      <update_component name=$client_id property="Read" value="true" />;
    let $activity_type := "Read";
  ?}
}
rule ClientWrite {
  input { $e : call_event; $io_id : string; $client_id : string; }
  output { $update_client : update_component; $activity_type : string; }
  trigger {? (contains($e/@method_name, "OutputStream.write") and $e/@callee_id = $io_id
?}
  action = {?
    let $update_client :=
     <update_component name=$client_id property="Write" value="true" />;
    let $activity_type := "Read";
  ?}
}
rule UpdateServer {
  input { string $server_id; string $activity_type; }
  output { update_component $update_server; }
  trigger {? ($activity_type = "Read") or ($activity_type = "Write") ?}
  action = {?
    let $update_server :=
      <update_componnet name=$server_id property="Activity" value=$activity_type />;
  ?}
}
composition System {
  CreateServer.$server_id -> ConnectClient.$server_id;
  ConnectClient.$client_id -> ClientIO.$client_id;
  ConnectClient.$client_id <-> ClientRead.$client_id;
  ClientIO.$io_id <-> ClientRead.$io_id;
```

```
  ConnectClient.$client_id <-> ClientWrite.$client_id;
  ClientIO.$io_id <-> ClientWrite.$io_id;
  ClientWrite.$activity_id -> UpdateServer.$activity_id;
  CreateServer.$server_id <-> UpdateServer.$server_id;
}
```

Figure 16. The DiscoSTEP program for mapping between a run of the program in Figure 15 and a simple client-server architecture.

```
PROGRAM ::=
    IMPORT*; EVENT; (COMPOSITION | RULE) *

IMPORT  ::=
  import quoted file name

EVENT::=                                           event type declarations:
      'event' '{'
        'input' '{' (ID ';')* '}'
        'output' '{' (ID ';')* '}'
      '}'

RULE  ::=                                          rule declarations:
      'rule' ID '{' RULEPARTS '}'

RULEPARTS[1] ::=                                    rule property declarations:
      'input' '{' (ID VARID ';')* '}'
      'output' '{' (ID VARID ';')* '}'
      'trigger' '{$' XPRED '$}'
      'action' '{$' XQUERY '$}'

COMPOSITION ::=                                    composition declarations:
      'composition' ID '{' COMPOSITIONPART* '}'

COMPOSITIONPART::=                                 composition property declarations:
      MEMBER '->' MEMBER
      MEMBER '<->' MEMBER

MEMBER ::=
      ID '.' VARID                  |
      ID '.' MEMBER

ID ::= [a-zA-Z][a-zA-Z0-9_]*

VARID ::= [$][a-zA-Z0-9_]*
```

Figure 17. The concrete syntax of DiscoSTEP.

[1] Note that the productions XPRED and XQUERY in the language refer to XQuery Predicates and XQuery FLWOR expressions, respectively. The grammar for these is defined in [XQuery].