DiscoTect: A System for Discovering the Architectures of Running Programs using Colored Petri Nets

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

March 2006
CMU-CS-06-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

One of the challenging problems for software developers is guaranteeing that a system as built is consistent with its architectural design. In this paper we describe a technique that uses run time observations about an executing system to construct an architectural view of the system. 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. 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.

The research described in this paper was supported by DARPA, under Grants N66001-99-2-8918 and F30602- 00-2-0616, by a Software Engineering Institute (SEI) Internal R&D Grant, and by the ARO under Grants DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") and DAAD19-01-1-0485. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies.

Keywords: Software Architecture, Dynamic Analysis, Colored-Petri Nets

Table of Contents

1 Introduction

A well-defined software architecture is critical for the success of complex software systems. An architectural model consists of many views (e.g., [24]) of the system. One particularly useful view is the component and connector (C&C) view, which 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) [5][32][36]. As an abstract representation of a system, such 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 [7][8][27].

However, 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.

Two techniques have been used to determine or enforce relationships between a system's C&C 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 [2]) where 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 [35][38][39].

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 [17][21][28]. 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 [8][14]. 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 using static analysis 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 temporally interleaved with many other architecturally relevant actions. Moreover, there is likely no single architectural interpretation that will apply to all systems: different systems will use different runtime patterns to achieve the same architectural effect and, conversely, there are many possible architectural elements to which one might map the same low-level events.

In this paper, we describe a technique to solve the problem of dynamic architectural discovery for a large class of systems. The key idea is to provide a framework that allows one to map implementation styles to architecture styles. This mapping is defined conceptually as a Colored Petri Net [19] that is 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 way to identify when a program performs "architecturally significant" actions that produce architectural structures. An important additional feature of the approach is the ability to reuse such mappings across systems. In particular, we exploit regularity in 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 cost of writing each abstraction mapping, while still providing flexibility.

In the remainder of this paper, we describe the approach in detail, and describe the tool called *DiscoTect*, that we have implemented. Section 2 presents the DiscoTect approach, including an overview of the DiscoTect framework, and the DiscoSTEP language used for specifying mappings. We then outline a formal semantics for DiscoSTEP that specifies its meaning in terms of Colored Petri Nets. We illustrate this with a simple example. In Section 3 we describe the implementation of DiscoTect. Sections 4 and 5 present case studies that we use to evaluate the efficacy of DiscoTect. In Section 6 we position our work in the context of related research. Finally, in Section 7 we present our conclusions and future work.

2 DiscoTect

2.1 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. At any 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, 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. To handle this variability of implementation strategies and possible architectural styles of interest, a language is required to define new mappings. Given a set of implementation conventions (which we will refer to as an *implementation style*) and a vocabulary of architectural element types and operations (which we will refer to as an *architectural style* [12]), we require a description that captures the way in which runtime events should be interpreted as operations on elements of the architectural style. Thus each pair of implementation style and architectural style has its own mapping. A significant consequence is that these mappings can be reused across programs that are implemented in the same style.

In summary, taking these challenges into account produces the following requirements for an approach to dynamically determine the architecture of a running system:

1. It must handle M-N mappings between low-level system events and constructs at the architectural level;
2. It must be able to keep track of concurrent and interleaved sets of event traces that produce architecturally significant events;
3. It needs to have flexibility in mapping implementation style to architectural style.

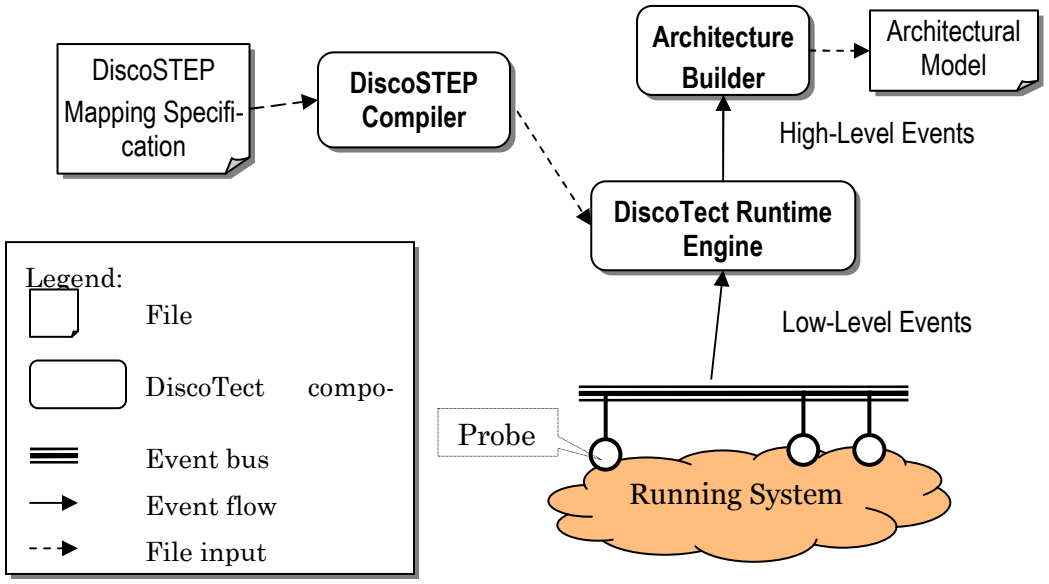

Figure 1. The DiscoTect Architecture

2.2 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 Runtime 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 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:

1. **DiscoSTEP Mapping Specification.** We have developed a language, called DiscoSTEP for specificying mappings between low-level and architecture events. The execution semantics 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 Section 2.4.
2. **DiscoTect Runtime Engine.** The run-time engine takes, as inputs, events from the running program and a DiscoSTEP specification and then 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.

2.3 Informal Introduction to DiscoSTEP

To handle the variability of implementation strategies and architectural styles of interest, we provide a language to define mappings. DiscoSTEP specifies how to map 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 that can be interpreted by the DiscoTect engine. The DiscoTect engine processes the runtime events produced by the 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.** Events are the kinds of events that are consumed and produced by the Disco-STEP program. These are defined by an XML Schema.
2. **Rules.** Rules specify how to map a set of system-level events into architectural events, and represent a single "step" in processing an event stream.
3. **Compositions.** Compositions of rules specify how output events from rules are passed as input events to other rules. Complex patterns of event sequences to be processed can be defined in this way.

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.

2.3.1 Events

In DiscoTect we capture *runtime events* such as method calls, CPU utilization, network bandwidth consumption, memory usage, etc. To generate these events at runtime, we probe, or instrument, the running system. To do this, we can use resource monitoring tools, or code instrumention tools such as AspectJ [23] and AspectC++ [37] that allow us to inject code into the target system. These events are input into a DiscoSTEP specification. The specification produces *architectural events*, generated as a result of processing the runtime events, which in turn are used to produce the software architecture.

2.3.1.1 Representation

```
<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.

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

Figure 3. An example "call" event.

We do not want the kinds of events consumed or produced by DiscoTect to be hard-wired. We want the flexibility of customizing events for particular implementation and architectural styles. To do this, we use XML Schemas to specify the valid XML representations that can be used as events by DiscoSTEP, and the events are therefore represented in XML. This allows us to perform structural type checking of the use of events in the DiscoSTEP specification. Figure 2 illustrates the schema definition of a call type event, and Figure 3 provides an example of a valid XML event that conforms to this schema.

2.3.1.2 Input and Output

Our event processing engine takes runtime events as input and produce architectural events as output. For clarity, and for the purposes of checking the correctness of a DiscoSTEP specification, we partition DiscoSTEP event types into *inputs* and *outputs* For example, the call type event described above would be specified as an input, whereas an event like "create_component" would be specified as an output event.

2.3.2 Rules and Compositions

Rules determine what events should be processed, and generate new events that are either used to construct the architecture, or are fed into other rules to allow recognition of complex implementation patterns. Rules are composed of *inputs*, *outputs*, *triggers* and *actions*. Input and output blocks declare input events that a rule cares about and the output events that a rule can 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 use a well-defined XML Query language called XQuery [45] to describe triggers and actions. This allows us to specify a large range of predicates and manipulations over XML, rather than inventing a new syntax.

A rule only represents a fragment of what is needed to capture an architecture. 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. In the simplest case, a rule consumes an event. However, in many cases we want the rule to be continually active; for example, a rule for recognizing a port on a component needs to fire multiple times so that multiple ports on the same component can be recognized. For this case, DiscoSTEP provides a bidirectional binding (denoted by <->). This means that a rule fires on an input without consuming that input (the event is implicitly reproduced as an output). Bidirectional bindings are a convenient shorthand for two directional bindings.

Section 3.4.4 provides concrete examples of rules and compositions.

2.3.3 Informal Runtime Semantics

Informally, DiscoTect runs DiscoSTEP specifications in the following sequence:

1. When an event is received by DiscoTect, associate the event with any rule that accepts 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 inputs to those rules. For output events 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 events to be used in the rules. Formally, we model an event as a colored tokens (where the color is given by the type), and a rule as a transition. For each event, a token is generated and put in place before a transition. Rules consume tokens and put them in places after transitions. We discuss this formal definition in Section 2.4.

2.3.4 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 type ClientThread) is started, which forwards all messages from that client to all connected clients.

While this system is simple, it allows us to illustrate the concepts of DiscoSTEP. In later sections we provide more complex case studies. In the remainder of this section, we show 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 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.

2.3.4.1 Instrumentation

The act of instrumenting a system to produce runtime events is not a novel aspect of DiscoTect. In fact, wherever possible, we use off-the-shelf technologies for our instrumentation. For Java-based systems we have used AspectJ to define instrumentation *aspects* that are weaved into the compiled bytecode of the 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.

2.3.4.2 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 Figure 13 of Appendix A, along with the source code of this example. These events can be fed into DiscoTect either in real time or off-line, after the program has completed running.

2.3.4.3 DiscoSTEP Program

A DiscoSTEP program that specifies how to handle the interleaved events between the client and server was specified to capture how to map system events into architectural events. The full specification is given in Figure 14 of Appendix A; 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 4 shows a fragment of the DiscoSTEP program that includes two rules, and how they are composed. The CreateServer rule constructs an architecture Server

```
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="{$client_id}" />;
    ?}
}
composition {
    CreateServer.$server_id -> ConnectClient.$server_id;
    …
}
```

Figure 4. The DiscoSTEP rule to create a Server component.

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.

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 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 the client com-

ponent 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).

2.3.5 Satisfying the Requirements

In this section we revisit the requirements for DiscoSTEP that we introduced in Section 2.1 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 data structures that can be passed from one rule to the next. These data structures are used to store information that can be passed between rules. Compositions define how this data is passed between rules.
- *Cope with concurrent states.* The informal execution semantics defined in Section 2.3.3 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 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 information can be managed.
- *Be flexible with respect to mapping implementation style to architectural style* The fact that the types of events consumed an produced by are specified in a DiscoSTEP mapping means that multiple implementation styles and architectural styles can be manipulated by DiscoTect. Furthermore, 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 a mapping between system file usage and 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 sum, meeting these requirements means that a DiscoSTEP mapping captures the way in which runtime events following an implementation style should be interpreted as operations on elements of an architectural style.

2.4 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 [19]. In [47] 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 to model the concurrency in the model. We believe that Colored Petri Nets are a more appropriate formalism for describing the semantics of DiscoSTEP mappings because their tokens provide a rich way of representing concurrent system states.

The DiscoSTEP language is formally described and modeled so that type checking and consistency checking can be done at the language level. The more interesting definition comes from how DiscoSTEP is formally modeled using Colored Petri Nets. Informally, we translate the DiscoSTEP constructs into CP-net constructs in the following way:

- The types for the CP-net are formed out of the union of event types used as input and output from all the rules used in a DiscoSTEP composition.
- Each DiscoSTEP rule becomes a CP-net transition
- Each group of connected input and output events becomes a CP-net place.
- Each DiscoSTEP trigger becomes a guard in the CP-net.
- The color for a place is derived from the connected input and output event type used to construct that place, which are guaranteed to be equivalent because of the DiscoSTEP type semantics.
- Each DiscoSTEP action becomes an action in the CP-net.
- CP-net arcs and nodes are constructed from the composition of the places and rules.

We continue 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.

2.4.1 DiscoSTEP Abstract Syntax

The concrete syntax for DiscoSTEP, which we have been using up to this point, is given in Figure 15 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 defined to be (@ 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 $\overline{v_{out}}$. Types include XQuery types as well as event types; we do not model type structure explicitly.

We do not directly model the semantics of XQuery, as they are defined elsewhere [45]. We also assume that all variable and rule names are globally unique.

2.4.2 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 well-defined runtime behavior.

Figure 5 shows the typechecking rules for DiscoSTEP. We briefly review the format for readers less familiar with the standard notation we use [31]. 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 $\Gamma \vdash C\ ok$ stating that a

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

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

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

Figure 5. The full set of type inference rules for DiscoSTEP.

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 6. The definition of Colored Petri Nets, from [19].

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 notation v:T means variable v has type T.

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.

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 Γ© eÀ: T stating that XQuery expression e has type T given assumptions Γ [45].

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.

2.4.3 Translational Semantics of the DiscoSTEP

According to [19], a CP-net has the definition presented in Figure 6. The translation semantics of the DiscoSTEP language define how to convert a DiscoSTEP program into a CP-net.

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

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

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

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

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

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

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

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

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

Figure 7. The Translational Functions for Mapping between DiscoSTEP and

Figure 7 gives the full set of translational semantics for mapping between Disco-STEP 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 7 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 4 the function GetType(CreateServer) returns the colors init, create_component, and string.

The main complication in the rules is the need to create a place for each group of connected input and output variables. We compute the set of places with the GetPlace function, which uses a union-find data structure. We use set notation to describe the initial state of the data structure, where we have a set of nodes $v_{in} \cup v_{out}$ each of which is its own equivalence class representative. The union operation $[v_1 \mathbin{B} v_2]u$ unifies nodes v_1 and v_2 in the union-find data structure u. We can merge two union-find data structures with the operator â as long as they have no nodes in common; if they do have nodes in common, the operation is undefined (we can always avoid this problem by naming variables uniquely in the source program, e.g. by using rule names as qualifiers). 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 $v_{in}::r$.

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 below. We get the set of places P as the equivalence class representatives in the union find data structure (i.e. those nodes for which $UF.find(x) = x$). We compute a map M mapping each variable to its equivalence class representative. Finally, this map is used to replace variable names with the canonical place name in the output of each helper function. The map operation does replace names inside of concatenations of the form $v_{in}::r$.

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

2.4.4 Formally Modeling the Example

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

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

The transitions in the CP-net are created as:

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

By applying other translations, the inputs and outputs are translated into CP-net places, arcs and node functions, 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 8 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 in the translational semantics, because the $client_id output of ConnectClient is bound to the inputs of more than one rule (ClientIO, ClientRead, and ClientWrite).

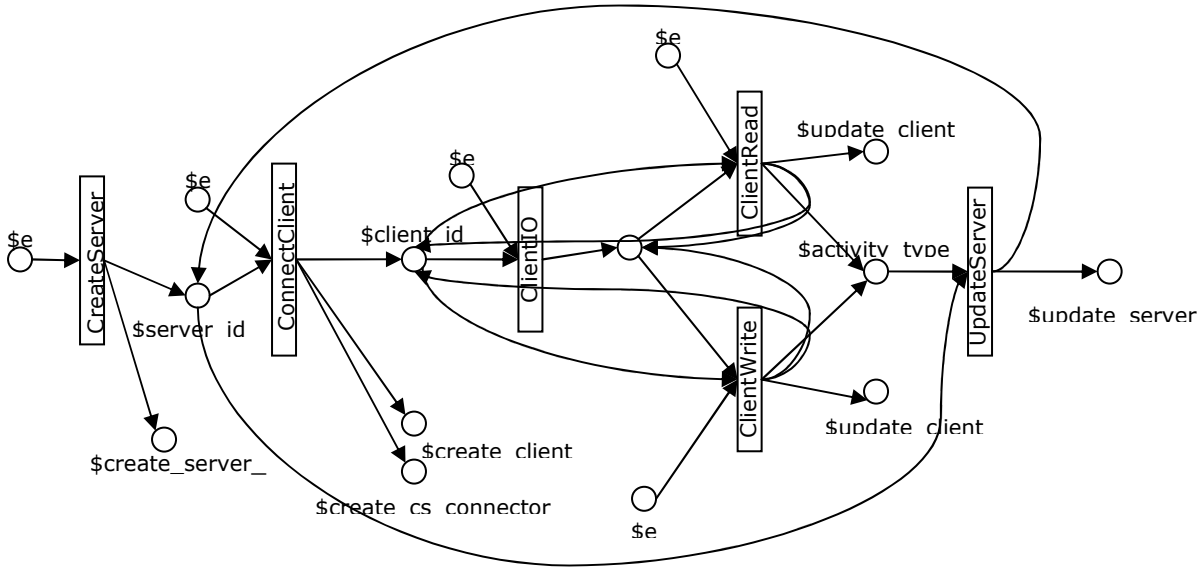

Figure 8. CP-net Places, Arcs and Node functions translated from inputs and outputs.

This CP-net representation is then "executed" in our toolset. When an event comes into the net, a token is created for each place that is relevant for that event, so that it can be compared with the associated guards for interested transitions. There is no priority on competing transitions – each matching set of tokens is processed by the transition when they match.

3 Implementation of DiscoTect

Recall from Section 2 that, to provide a framework for discovering architectures, we need to solve three challenges. In this section, we discuss our response to each challenge.

Monitoring: We use existing probing technologies to extract monitoring events. In the case studies in Sections 4 and 5 we used AspectJ to monitor 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: We have implemented the DiscoTect engine in Java, which follows the implementation outlined in this paper. The DiscoSTEP language is implemented using JavaCC, a Java-based compiler generator, and uses the Saxon XQuery processor to parse and execute the XQuery triggers and rules in a DiscoSTEP specification.

Architecture Building: We represent architectures using the Acme architecture description language [13] (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. The analysis capabilities of AcmeStudio [34] can then be used to check the architecture with respect to its style, or conduct analyses such as performance or schedulability.

4 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 [22]. The test-bed allows users to specify system resources, tasks and scheduling strategies, and simulates the running of the mobile system. We chose AAMS because it represents a fairly complex application (approximately 28KLOC), and the runtime architectural view of the system is well documented. This allows us to compare our discovery result with the original documentation. This comparison illustrates the use of applying our technique to discover deviations between the architecture discovered by DiscoTect and the documented AAMS architecture. Furthermore, this case study illustrates how we developed and refined the DiscoSTEP specification to produce the final architecture.

Figure 9 shows the runtime architecture of AAMS as presented in [22]; the following description of the runtime architecture is based on the description in this paper. The Simulation Controller forms a simulation from a description of resources

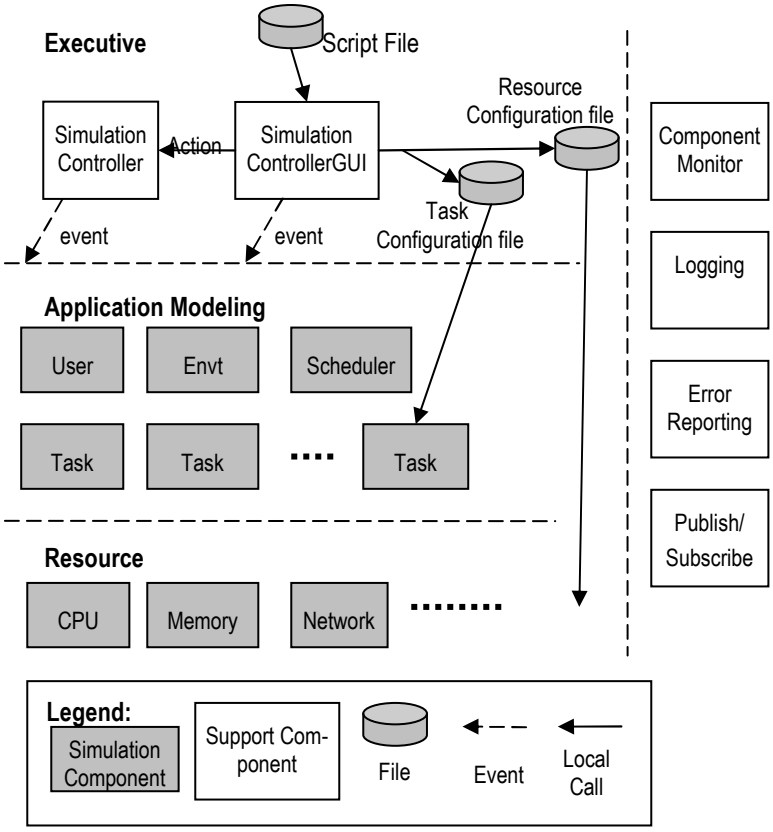

Figure 9. Documented runtime view of AAMS

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 running of the system. Each component in the application and resource layers simulates its own operation. A set of services for File I/O, Error Reporting and Logging are available via publish/subscribe to any simulated object.

4.1 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 an iterative process, starting with 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 component types are based on the description of AAMS found in [22].

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 itself. 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 specification for this portion of the architecture. This specification 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.

4.2 The Discovered Architecture

Applying the above DiscoSTEP specification to a running instance of AAMS yields the architectural model in Figure 10. (We have manually laid out this model to enable easier comparison with the view in Figure 9.) 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 rest of the system. This indicates that one instance is instantiated but never used. Code optimization should resolve this discrepancy.
2. *Additional connections between components*. Figure 10 does not show any connections between the controller component and simulation compo-

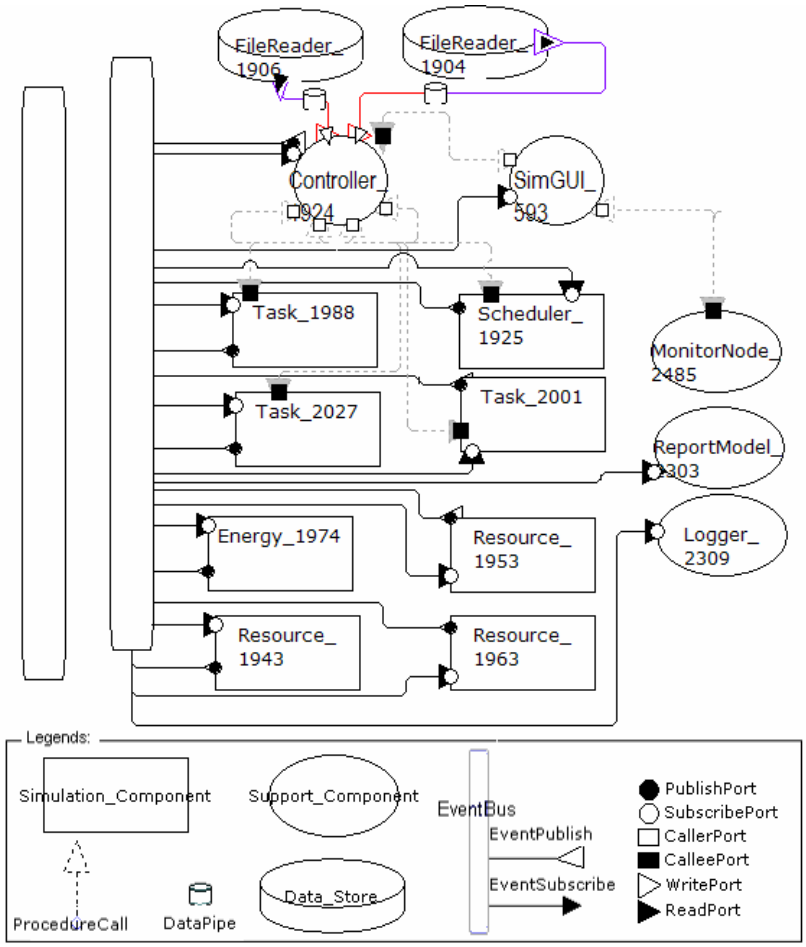

Figure 10. The Discovered Architecture of AAMS.

nents such as tasks and schedulers. Nor does it inform us that some of the support components (e.g. 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 or throughput analyses.

3. *Misplaced connections between components*. The discovered architecture shows a different File I/O scheme: instead of the GUI reading three files (c.f. Figure 9), 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.

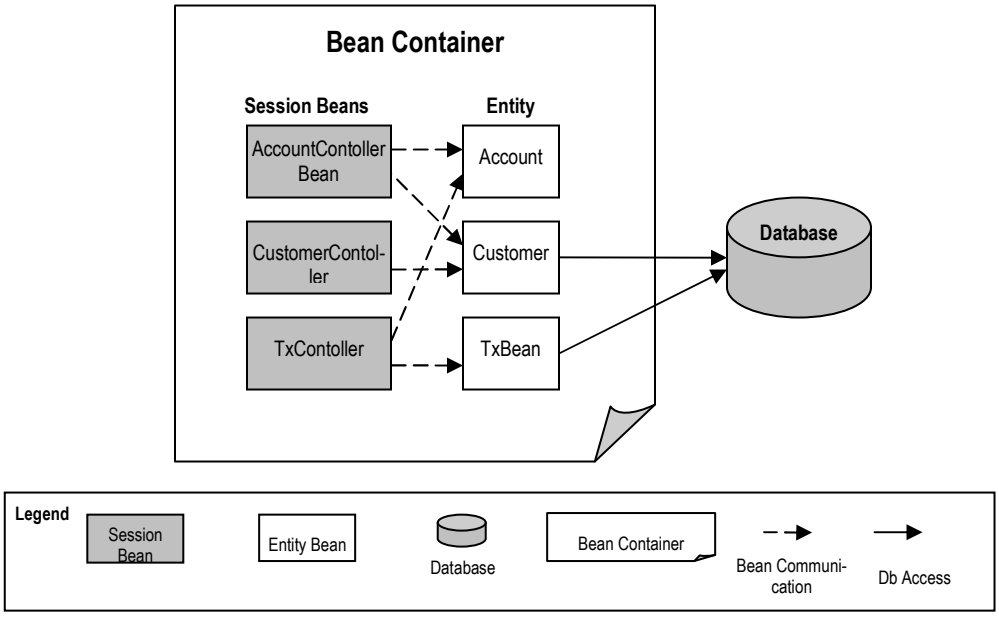

Figure 11. Documented architectural view of Duke's Bank Application

4. *Missing components/connectors*. Two of the components—User and Envt (Environment)—recorded in the document do not show up in the discovered 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 uncovered some errors in their coding. However, the missing User and Environment components are due to the fact that these represent user interaction, and are not actual components in the implementation.

5 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 Java Beans) banking application created by Sun Microsystems as a demonstration of EJB functionality. Duke's Bank allows bank customers to access their account information and to 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 [16], which enables us to compare the actual discovered architecture with the one presented in the documentation. For the case study, we used version 1.3 of the application provided by Sun.

We wrote an aspect that injected advice to object instantiations, method calls and field modifications. We compiled the Duke's Bank application along with the as-

pect, using an AspectJ compiler, so that system execution events were traced as the application ran.

Figure 11 gives a view of how the components interact in the Duke's Bank system as presented in [16]. 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.

5.1 Design of the EJB DiscoSTEP Program

We now present the steps taken to produce the DiscoSTEP specification to discover the Duke's Bank architecture. 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 [10].

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) [18], 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.

5.2 The Discovered Architecture

Applying the specification just described to a running instance of Duke's Bank yields the architectural model in Figure 12. We have manually organized the layout this model for better comprehensibility. We can make the following observations based on this process.

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

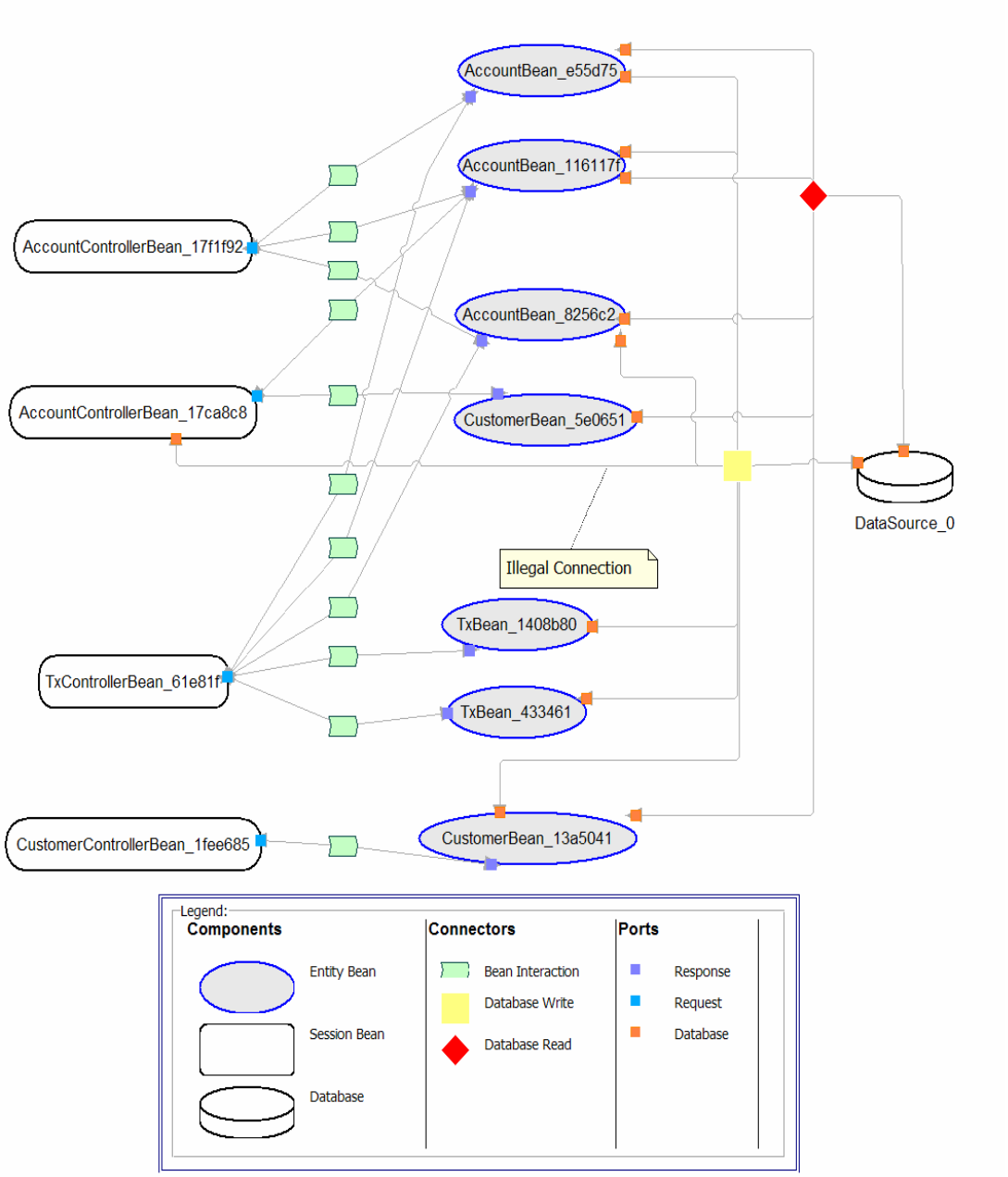

Figure 12. Discovered architecture of Duke's Bank

and container instances is useful for 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 12 are consistent with those described in the architecture shown in Figure 11: there are communication channels between AccountControllerBean and AccountBean, AccountControllerBean and Cus-

tomerBean, CustomerControllerBean and CustomerBean, TxControllerBean and TxBean, TxControllerBean and AccountBean.

3. *Discrepancies in Database Access*. Figure 11 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 [18] However a "database write" connector did appear in the discovered architecture. Further (manual) source code analysis confirmed that AccountControllerBean does directly write to the database; a violation of the architecture. As discussed in the previous section, identifying communication "backdoor" connections like this is useful for architectural analysis and to ensure architectural conformance.[1]

6 Related Work

To analyze a running system, it is first necessary to get information from it. There are many technologies available for monitoring systems, and we build on those. For example, Balzer [4] and Wells [44] provide systems to instrument the source code to produce trace information and manipulate runtime artifacts to get the information; aspect-oriented systems such as AspectJ allow use to instrument binary code when the source is not available We can use any of these approaches to get information out of the system.

Monitoring mechanisms do not by themselves solve the hard problem of mapping from code to more abstract models; event correlation and abstraction is also needed. Work by Dias and Richardson [9] uses an XML-based language to describe runtime events and use patterns to map these events into high-level events. However, analyzing these abstracted 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. Approaches to correlating events, for example GEM [29] and SEL [51] and work by Kaiser [20] provide sophisticated event correlation languages that can detect when certain combinations of events have occurred and, in some cases, what actions to perform based on those correlations. Our approach is similar, but it makes architectural styles or patterns explicit; our mappings can be used to map low-level events to high-level architectural events that are amenable to checking for consistency against particular architectural styles.

In prior work, we developed an infrastructure for doing certain kinds of abstraction [15]. However, this approach was limited to observing properties of a system and reflecting them in a pre-constructed architectural model. Here we show how to create such a model.

[1] Note that this error has been corrected in the most recent version of the Dukes Bank J2EE application.

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 [33][41][42][50]. 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 [11]. This is somewhat different from detecting architectural structure.

Madhav [26] 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 [2].

There has been research in using Colored Petri Nets to dynamically determine software architectures from running systems using Colored Petri Nets. This work either focuses on a particular quality attribute (e.g., [1] for performance analysis), uses UML as the modeling language, which is very close to the actual code [49] or checks interactions on pre-known connections are of the correct protocol [6]. In contrast, our approach focuses on determining the component and connector architecture view of a running system where the architecture particular components and connections may not be known a priori, and where the actual architectural view may be much more abstract than the programming language constructs used to implement the system. Moreover, our approach is agnostic about the specific architectural styles that may be used and representation language.

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 [3] and Vieira and colleagues [40]). These approaches, however, do not link architecture to an executing system.

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 concurrent 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 here, we have been able to discover the Pipe/Filter architecture of systems 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 [21] or Armin [30] 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 mentioned, we have experimented with a number of existing monitoring technologies for Java and C++. However, monitoring technologies for other kinds of implementations and system properties is a 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 envision a large library of recognizers for common architectural frameworks available 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", with some certainty, 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.

Finally, we are still gaining experience with use of these tools in practice. The process of developing the DiscoSTEP mapping is a long, iterative process, that currently requires intimate knowledge of the implementation conventions. While we can never truly eliminate this need, we are investigating process, conventions, and tools, that could make this process less time-consuming and error-prone.

Acknowledgements

The research described in this paper was supported by DARPA, under Grants N66001-99-2-8918 and F30602- 00-2-0616, by a Software Engineering Institute (SEI) Internal R&D Grant, and by the ARO under Grants DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") and DAAD19-01-1-0485. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies.

References

[1] M. Ajmone Marsan, G. Balbo, G. Chiola, and G. Conte, "Modeling the Software Architecture of a Prototype Parallel Machine," Proc. SIGMETRICS Performance Evaluation Review, Vol 15. No. 1, pp. 175—185, 1987.

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

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

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

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

[6] Q. Bai, M. Zhang, and K.T. Win, "A Colored Petri Net Based Approach for Multi-agent Interactions", Proc. of 2nd International Conference on Autonomous Robots and Agents, Palmerston North, New Zealand, December 2004.

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

[8] 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, 2002.

[9] 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.

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

[11] 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.

[12] D. Garlan, R.J. Allen, and J. Ockerbloom. "Exploiting Style in Architectural Design," Proc. of ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, 1994.
[13] D. Garlan, R.T. Monroe, and D. Wile. "Acme: Architectural Description of Component-Based Systems," Foundations of Component-Based Systems, G. Leavens and M. Sitaraman (eds), Cambridge University Press, 2000.
[14] 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.
[15] 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.
[16] Sun Microsystems <http://java.sun.com/docs/books/j2eetutorial/index.html>
[17] D. Jackson and A. Waingold. "Lightweight extraction of object models from bytecode," In Proc. of the 1999 International Conference on Software Engineering, 1999.
[18] Sun Microsystems <http://java.sun.com/products/jdbc>
[19] 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.
[20] G. Kaiser, J. Parekh, P. Gross, and G. Veletto. "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems," Proc. of 5th International Active Middleware Workshop, 2003.
[21] R. Kazman, and S.J. Carriere. "Playing Detective: Reconstructing Software Architecture from Available Evidence," Journal of Automated Software Engineering 6(2), 1999.
[22] R. Kazman, J. Asundi, J.S. Kim, and B. Sethananda. "A Simulation Testbed for Mobile Adaptive Architectures," Computer Standards and Interfaces, 2003.
[23] G. Kiczales, E. Hilsdale, J. Huginin, M. Kersten, J. Palm, W. Griswold. "Getting Started with AspectJ," Communications of the ACM 4(10), October 2001.
[24] P. Kruchten. "The 4+1 View Model of Architecture." IEEE Software, 12(6):42-50, 1995.
[25] D.C. Luckham. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," Proc. of DIMACS Partial Order Methods Workshop, 1996.
[26] N Madhav. "Testing Ada 95 Programs for Conformance to Rapide Architectures," Proc. of Reliable Software Technologies – Ada Europe 96, 1996.
[27] 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.
[28] G.C. Murphy, D. Notkin, and K.J. Sullivan. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," In Proc. of 1995 ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995.
[29] M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems". IEE/IOP/BCS Distributed Systems Engineering Journal Vol. 4, No. 2, June 1997.
[30] L. O'Brien, C. Stoermer, "Architecture Reconstruction Case Study," Software Engineering Institute Technical Note CMU/SEI-2003-TN-008, 2003.
[31] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[32] D. Perry, A. Wolf. "Foundations for the Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, 17(4), 1992.
[33] S. Reiss. "JIVE: Visualizing Java in Action (Demonstration Description)," Proc. of 25th International Conference on Software Engineering, 2003.
[34] B. Schmerl, and D. Garlan. "AcmeStudio: Supporting Style-Centered Architecture Development (Demonstration Description)," Proc. of the 26th International Conference on Software Engineering, Edinburgh, Scotland, May 23-28, 2004.

[35] 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.

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

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

[38] 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.

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

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

[41] 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. of OOPSLA'98,

[42] 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. of CASCON 2000.

[43] 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/.

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

[45] W3C http://www.w3.org/TR/xquery/

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

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

[48] H. Yan, J. Aldrich, D. Garlan, R. Kazman, and B. Schmerl, "Discovering Architectures from Running Systems: Lessons Learned," Technical Report CMU/SEI-2004-TR-016, December, 2004.

[49] J. Xu and J. Kuusela. "Modeling Execution Architecture of Software System Using Colored Petri Nets," Proc. of First International Workshop on Software and Performance (WOSP'98), Santa Fe, NM, pp. 70-75, October 1998

[50] A. Zeller. "Animating Data Structures in DDD," In Proc. of SIGCSE/SIGCUE Program Visualization Workshop, 2000.

[51] D. Zhu and A.S. Sethi, "SEL, A New Event Pattern Specification Language for Event Correlation", Proc. ICCCN-2001, 10th International Conference on Computer Communications and Networks, Scottsdale, AZ, Oct. 2001.

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 13. The Java 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 14. The DiscoSTEP program for mapping between a run of the program in Figure 13 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[2] ::=                                       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 15. The concrete syntax of DiscoSTEP.

[2] 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].