

An (Architecture-centric) Approach for Tracing, Organizing, and Understanding Events in Event-based Software Architectures

Scott A. Hendrickson, Eric M. Dashofy, and Richard N. Taylor

*Institute for Software Research, University of California, Irvine
Irvine, CA 92697-3425*

+1 949 824 4101

{shendric, edashofy, taylor}@ics.uci.edu

Abstract

Applications built in a strongly decoupled, event-based interaction style have many commendable characteristics, including ease of dynamic configuration, accommodation of platform heterogeneity, and ease of distribution over a network. It is not always easy, however, to humanly grasp the dynamic behavior of such applications, since many threads are active and events are asynchronously (and profusely) transmitted. This paper presents a novel, complete approach that aids in the understanding, debugging, and visualization of the behaviors of event-based applications. It applies to real, implemented systems, without requiring the presence of component source code, and supports partial or incomplete, heuristic behavior specifications. A prototype implementation of our approach was applied to two systems, including the prototype itself, indicating that our approach is feasible, scalable, and shows promising results in terms of increasing the understandability of these types of systems.

1. Introduction

Event-based architectural styles are styles in which software components communicate with each other via explicit software connectors using *events*, or *messages*. Each component runs in its own memory space with its own thread(s) of control. Events are discrete data objects that are not allowed to contain direct pointers to data in memory or control entities like thread objects. Because there is no assumption of a global clock or ordering of execution among components, event-based systems are *asynchronous*—a component may send an event at any time, and may receive an event at any time. Systems built in this manner have many beneficial characteristics such as low coupling, ease of dynamic reconfiguration and ease of distribution across multiple heterogeneous platforms.

Understanding, debugging, and visualizing the behaviors of event-based systems is a complex task for many reasons. Some difficulties are shared with traditional procedural or object-oriented systems, such as the use of off-the-shelf components written by third

parties, provided without source code. Others are unique to asynchronous event-based systems. Large event-based systems can quickly generate millions of events. Furthermore, because events can be processed in different orders, and because each component can run in a separate thread of control, it can be extremely difficult to understand the relationship of one event to another, or a set/sequence of events to a particular observed behavior.

The differences between event-based and traditional systems have both positive and negative consequences for understanding, debugging, and visualizing program behaviors. On one hand, event-based systems are in a sense more transparent than tightly-coupled systems, since all communication is reified in the form of events. On the other hand, the quantity of events and their non-obvious relationships to corresponding behaviors is a major inhibitor to understanding.

The contribution of this paper is a novel, complete approach that aids in understanding, debugging, and visualizing the behaviors of event-based applications. The approach has four main parts:

1. **Event capture:** the implemented system is instrumented and executed. During execution, events are captured and stored in a database. Events are linked to the system through the use of a structural model of the communication pathways in the system.
2. **Event classification:** based on available knowledge of event content, users write classifiers that categorize each event into one or more semantically meaningful categories. This aids in sorting, filtering, and determining causal relationships.
3. **Event causality:** based on available knowledge about the behavior of individual components, users write rules that describe how components react to certain inputs. This aids in event visualization by helping the user understand relationships between events. Unlike other approaches, these rules are optional and heuristic: while rules do not need to be present or complete, more accurate rules result in more accurate visualizations.
4. **Event visualization:** Data in the event log is pre-

sented to the user in an organized, browseable manner. Multiple views can be employed, from simple tabular views to views that incorporate graphical models of the system that generated the events. While simply organizing this mass of data provides tremendous value, visualization become especially effective when the rules written in the previous steps are leveraged by these views to guide the user in gaining an understanding of event-driven interactions among components.

This approach is novel for several reasons. It is applied to implemented, running systems rather than simply abstract specifications of such systems, so the event data and system models used are accurate with respect to the real system. It does not require the presence of source code, since event data is used as the primary source of system information. Most importantly, it is incremental and heuristic: once events are captured, visualizations can provide value even if the user writes no rules. The value of the visualizations increases as the user writes more (or more accurate) rules describing component behavior.

We have evaluated our approach by implementing a prototype for a particular Java event-based middleware called C2.FW. For **event capture**, we implemented a tool called the INSTRUMENTER that automatically inserts tracing connectors into a C2.FW system that capture run-time events. For **event classification**, we developed a mechanism for performing dynamic queries over the captured event log. For **event causality**, we developed two semantic classes of rules to specify causal relationships between system events. For **event visualization**, we built a tool, MTAT, which combines the event log with classifiers and rules, allowing the user to browse the event log, view events in terms of a graphical visualization of the systems we analyzed, and explore causal relationships between events. We applied our prototype to two previously-developed C2.FW systems: KLAX, a video game, and an AWACS (Airborne Warning and Control System) aircraft software system simulator.

Overall, our experience implementing our approach indicates that it is feasible, scalable, and shows promising results in terms of increasing the understandability of the systems to which we applied it. We believe that this approach can be implemented for different classes of event-based middleware in different settings, and augmented with additional visualizations.

2. Approach

While traditional approaches to understanding, debugging, and visualizing program behaviors may work well within a single component of an event-based system, they are not equipped to deal with the complex interactions that occur *among* components. They do not take into account the unique characteris-

tics of these interactions, such as asynchrony, extensive concurrency, the use of events rather than procedure calls, and the use of loosely-coupled, often black-box components.

In contrast, our approach leverages the fact that communication in these systems is reified as distinct events and is relayed by explicit software connectors, such as those implemented by message-oriented middleware frameworks. It then provides techniques for organizing and making sense of the vast quantity of events that can be produced in a system run, and combines this with structural information about the system to provide meaningful visualizations. The approach uses heuristic mechanisms because information and knowledge about such systems is often partial or incorrect.

We do not believe that this approach is, in isolation, a ‘silver bullet’ for understanding the behaviors of event-based systems. Rather, it should be used in combination with other approaches for maximum effectiveness.

2.1. Capturing events

Understanding, debugging, and visualizing program behavior requires information about the state of a running system over time. Traditional program debuggers rely on source code and gather information about the state of the system by providing an instrumented, controlled environment in which software programs run. This is necessary largely because communication among modules and procedures in these systems is implicit, via procedure calls, and system state is stored in inaccessible variables.

Event-based systems provide an alternative source of information because they reify all inter-component communication in the form of *events*, or *messages*. Events are independent, self-contained data structures that do not contain direct pointers back to data residing within a component or control objects like threads. Thus, data in an event can generally be interpreted without knowledge of some component’s state.

Our approach uses event capture as the primary source of information about an event-based system for several reasons:

1. Events pass through explicit software connectors, usually in the form of middleware or an event-based framework making instrumentation for the purpose of event monitoring feasible in a variety of contexts.
2. Event monitoring does not require the presence of component source code, so it is amenable to systems that include black-box components (as are common in loosely-coupled, event-based systems).
3. Event monitoring gives a complete, accurate record of actual component interaction without simulations, formal specifications, or other sources of information about the system.

In our approach, no particular method of event capture is mandated. There are many possible ways to capture events from an event-based system, depending on its environment. Some examples include:

Middleware: Most component-based message-passing systems are built using some sort of message-oriented middleware or architecture framework. Many include a monitoring or logging interface that can be used to directly capture the events. This approach works well when such an interface is available. Furthermore, instrumentation technology need only be implemented once per middleware/framework.

Add instrumenting elements: Many event-based systems deploy or connect their components at system-instantiation time, guided by some deployment descriptor or other structural description document. In these cases, it is possible to modify the structure of the application itself by inserting specialized, first class entities that intercept and capture each event.

Use existing instrumentation: It may be possible to leverage existing logs as a source of event information. Depending on the inherent logging capabilities of the system in question, the logs may not fully capture all events, but our approach can generally be applied even without totally complete event traces.

Regardless of the source used, our approach imposes one constraint on the captured events: it must be possible to relate the events back to two structural elements in the system: the component (or connector) that sent the event and the element that received it. Otherwise, it is not possible to characterize interactions among components using captured events, nor relate events to one another in terms of causality (a key part of our approach discussed later in this section).

In the prototype implementation of our approach, we chose to add instrumenting elements, called trace connectors, to a system's deployment description as specified in the xADL 2.0 architecture description language. These trace elements capture all events sent between components and connectors in the system, parse out content from events with known formats, and log the data (including sender/receiver) in a relational database. The use of a relational database is not a required part of our approach, but we found that their ability to quickly store and retrieve large amounts of data, as well as their highly-optimized query capabilities, were critical to the success of our implementation.

2.2. Event classification

Event-based systems generally organize data within an event according to some sort of structure. Examples include a name-value pair table, an XML tree, a node graph, or a partitioned binary structure as might be found in an IP packet. Complex systems may use more than one format.

Making sense of a collection of captured events requires parsing the data contained within them and

allowing the user to classify events according to different criteria. This step can be thought of as applying different queries over the entire event trace, or a subset thereof. Classification of events may be manual, initiated by the user, or automatic, initiated by a tool to find events with certain characteristics.

Events can be classified in order to distinguish them in visualizations; for example, one might want to highlight all events from a particular component using a user-chosen color, or perhaps exclude them from view when the user has determined that they are unimportant for a particular purpose. Events can also be classified based on their place in a particular behavior or causal chain, as will be discussed below.

Once events are classified, it is possible to use different types of tools over the symbolic event classifications to detect potential causal relationships or to bolster or weaken a belief in other causal relationships.

It is important that the assignment of each event into classes be dynamically determined after a trace has been captured, rather than assigned during a trace. This accommodates incrementally refining, adding, and removing classifiers as event purposes are discovered, verified, or refuted.

In our prototype implementation, we have developed a simple property-based method of writing classifiers (essentially queries) that balances expressibility and simplicity. Finding events that match a given classifier is done using the query capabilities of the relational database event store. Event classifiers are used for a number of purposes in our prototypes: to help the user find specific events, to allow the user to highlight events on, or exclude events from a visual display (allowing divide-and-conquer exploration of the event log), and for writing causality rules (see next section).

2.3. Event causality

Understanding and debugging the behavior of a software system is usually a matter of finding a causal chain between an input and an observed output. In traditional procedural system, the causal chain among procedures and modules is relatively straightforward since control flow follows a particular thread of execution. When a procedure call occurs, control flow passes to the called procedure and returns to the caller when the procedure terminates.

In an event-based system, things are not so simple. Asynchronous components are free to receive and send events at any time. In general, an event-based component will receive a set of input events and react at some later time by emitting events in response to these inputs. However, without access to, or complete knowledge of, a component's source code, it is often difficult to determine which events were emitted by a component in response to which input events. We believe that understanding the causal relationships between events received by a component and those emit-

ted in response is a key part of improving the understandability and debuggability of event-based systems.

We say that emitted events are *caused* by received events when an event (or set of events) must be received by a particular part of the system *before* it will emit an other event (or set of events) in response. Not all emitted events must have event causes—an event may be emitted in response to an environmental condition (a clock component emitting periodic ‘tick’ events, or a user interface component emitting ‘key-stroke’ events). Likewise, an event does not have to cause other events—a part of the system may absorb or ignore an event entirely.

In an ideal world, it would be possible to determine causality among events with absolute certainty, with no additional help from the user. This can be accomplished if components tag each emitted event with a list of ‘caused-by’ events or by leveraging a complete formal model of system behavior that makes it possible to determine the causality for any given event.

Realistically, however, we can not assume that component developers will be so cooperative or that such comprehensive formal models exist (or could be feasibly developed). In general, raw event logs do not contain enough information to deterministically identify causal relationships among events. Consequently, it is necessary for users to assist tools in identifying causal relationships between events.

Fortunately, improving system understanding and debugging does not require absolute certainty about causality. These activities can also be supported if *probable* causal relationships are available. Probable causality can often be determined using far simpler, incomplete models of component behavior. One benefit of this approach is that as models are refined, the probability of correct causal relationships improves.

In our approach, we advocate the development of behavioral *rules* that decorate components and connectors and describe aspects of their behavior. In our approach, a rule must specify at least two things:

1. A set of characteristics (i.e. classifiers) for stimulus, or *cause* events.
2. A set of characteristics (i.e. classifiers) for response, or *effect* events.

For example, a simple rule might indicate “when a `query_current_time` event is received by the `system_clock` component, it emits a `current_time` event.” Rules can be written using any number of sources of information about behaviors, from previously observed behaviors to component documentation to the source code of a component itself (if available). The specific semantics of rules are highly dependent on the content of messages and the query capability of the classification engine, as well as other considerations such as architectural style constraints. As such, it is not possible for us to advocate any particular language or formalism for expressing these rules, al-

though there are many available (e.g. LTL and other temporal logic languages).

The primary advantage of using (incomplete) behavioral rules over complete formal models is that they are heuristic. The better the behavioral annotations, the more accurate the relationships that should emerge. It may even be possible to determine absolute causality for particular kinds of events, such as those that share a unique key like a transaction identifier. With no rules, generic rules can be substituted and should still be able to produce crude information about causality. An obvious such general rule might be *post-hoc ergo proctor hoc* (happened after, therefore caused by), i.e. “the event(s) emitted by this component were caused by the most recently received event.”

In our prototype, we developed two classes of rules that were representative of the most common kinds of component behavior used in the event-based systems we examined. Specifics of these rule types will be discussed later in Section 3.3. Rules basically consist of ‘cause’ and ‘effect’ classifiers that are used to match events; an automated tool turns these rules into database queries and displays the results. In section 4.1, we discuss our approach to creating classifiers and rules within the context of the C2 event-based framework.

2.4. Visualizations

Classifying events and identifying causal relationships is a definite step forward in terms of allowing a user to explore event traces and system behaviors, but these mechanisms must be accompanied by visualizations that help to present this information to the user in a well-organized way. Minimally, visualizations should provide interactive access to the event log; however, event-based systems generate far too much information to be viewed all at once. Implementations of our approach must include one or more tools to visualize traced events, ideally providing information about the traced events and their relationships in different ways.

Tools should minimally provide access to the complete and correct log of application events and support customized filters that show, hide, or highlight events based on source, destination, interface, time, or other event property, as well as by event classifiers.

Causality information should be included in these visualizations by clearly indicating an event’s possible causes and effects using the behavioral rules defined for participating components. Generally, a single event is part of a larger causal chain, starting with an environmental input and proceeding through many cause-effect pairs through a resulting behavior. As a user steps through potential causal chains, these chains should be clearly identified and displayed to the user allowing a user to step back to any point in the chain and pursue different potential causal chains.

Finally, since causal relationships may be heuristic, visualizations should support a user in verifying whether a reported cause or effect is accurate. One way that this can be accomplished is by making available information about each message.

Event classifications and causal chains can be viewed in isolation; that is, without direct visualization in the context of the system structure. Indeed, Luckham proposes an event-centric visualization that presents events as nodes in a graph connected by causal links [14]. This approach can clearly show multiple causal chains between events, over any period of time, and at different levels of abstractions using aggregators. However, using this approach makes it difficult to see what parts of the system are doing the work, even when this information is textually present, since the structure of the system is missing.

Instead, we strongly advocate including viewpoints that take into account system structure. One effective visualization we have found superimposes event information atop a graphical depiction of the system structure by highlighting the link between appropriate parts of the system over which an event was captured. Of course, this requires a graphical model of system components, but this can often be obtained from design documents or deployment descriptors. Using a graphical system structure, when available, clarifies the context of individual events, greatly enhancing the user's ability to understand the control flow through different parts in the system. This visualization is not perfect, however: in this view it can be difficult to grasp a broader picture of the system and multiple causal chains due to graphical clutter.

Another alternative view is a communication-centric viewpoint, similar to UML sequence diagrams for object oriented systems. This view has some of the advantages of the previous two visualizations. It clearly shows the source and destination of events as well as multiple events over time, and does not require a graphical depiction of the system. This view is advantageous when viewing individual causal chains in large systems, since components and connectors that are not involved in the chain are elided.

In our prototype implementation, we chose to focus on supporting communication- and structure-centric approaches to visualization, as we feel these most strongly fit the goal of understanding the different parts of a system, and how they interact. Our visualizations allow the user to interactively explore the event log, use classifiers to query and highlight messages, define new classifiers and causality rules, and explore causality chains induced by those rules.

3. Implementation

Here, we introduce our prototype implementation of our approach, describing specific implementation-

level choices we made for each step in the process.

3.1. Event capture

The C2.FW/xADL 2.0 infrastructure affords us the ability to capture events by either instrumenting the framework, for which it exposes programming hooks, or to modify the system's deployment descriptor (as specified in xADL 2.0) to add instrumenting elements.

Although either one of these approaches would have worked, we chose to add trace elements, called trace connectors, into the system's architecture description for several reasons. First, adding trace connectors did not require us to change framework code. Second, if we want to trace events from only part of the system, we can insert trace connectors only on the links that we are interested in, rather than having to recode the framework implementation. Finally, we felt that this approach was generally more applicable than framework instrumentation, since it works for any architecture instantiated with a deployment description and is not dependent on any particular framework.

To instrument a system, we developed a prototype tool called the INSTRUMENTER that examines a xADL 2.0 architecture description of a system's structure and automatically inserts trace connectors and logging components into the description. Trace connectors intercept all messages passing through them and send a copy of each message to logging components that log them to a relational database.

A logger component connected to each trace connector inserts an unmodified binary copy of each message into the database along with generic properties about the traced message such as which component produced it, which received it, on what interfaces, and at what time. It also logs the process and message id produced by the trace connectors.

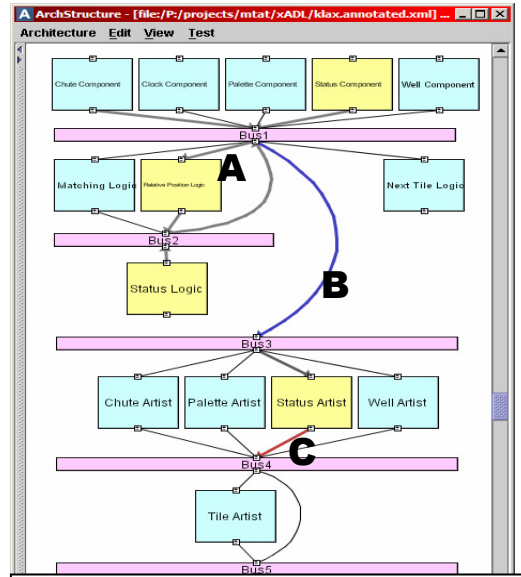
Most C2.FW events have a character string name and a set of name-value pair properties. The string values of names and properties of messages are stored alongside the binary copies in the database. Having access to the contents of messages is, of course, critical: our approach would be rendered useless if each message could only be treated as a binary blob.

3.2. Event classification

An *event classifier* is a test that can be run on events. Recall that event contents in our infrastructure consist of name-value pairs. Event classifiers consist of a set of 3-tuples, with each tuple containing a property name, an operator, and a property value. The operator and value specify a constraint an event's property must meet, or if omitted, indicate a required property without a constrained value. To match the classifier, an event must adhere to all of its constraints.

MTAT (below)
Provides access to the traced events, computes causality

- A) Supports a user in querying traced events
- B) Displays the results of the query in A
- C) Potential causes of an event using rules
- D) Potential effects of an event using rules
- E) Displays details of a selected event
- F) Emerging causality chain



- Archipelago (above)**
Displays the system structure, event links
- A) Highlighted link(s) of causally related events
 - B) Highlighted link(s) of currently listed causes
 - C) Highlighted link(s) of currently listed effects

Figure 1. MTAT and Archipelago screen shots.

The primary reason we chose this format for classifiers is because we believe it maintains a balance between simplicity and expressibility; different domains may of course implement different classifier semantics optimizing for one or the other. A secondary reason is that these classifiers can be straightforwardly transformed into SQL queries that can be applied to events in the relational database. As such, operators in classifiers are those available in SQL, e.g. greater-than, like/not-like, not-equal, and so on.

3.3. Determining causal relationships

Because applications in C2.FW do not tag messages with their causes, it is not possible to determine the causality relationships of messages with absolute certainty. Instead, we must write rules describing the behavior of components, which can be used by automated tools to determine causality.

For C2.FW applications, we have developed simple semantics for two kinds of behavioral rules. In this implementation of our approach, each rule defines a set of *causes* and a set of *effects*. Causes and effects are specified in terms of message classifiers, the same as those introduced in the last section. Additionally, rules include a number of required occurrences of the event type, and the component/connector interface on which the message would be received or emitted. Finally, a relationship between the property values of any of the specified causes and effects may be specified.

The two types of rules we have defined are *MatchingN*, and *MostRecent*. A *MostRecent* rule is used to indicate that a component *may* emit a message matching the ‘effect’ classifier when it receives an event matching the ‘cause’ classifier. A *MatchingN* rule is used to indicate that a component will *always* send the complete set of effect messages when it receives a complete set of cause messages. This rule type is especially applicable to components that queue up requests or that broadcast messages such as a FIFO bus connector (a common construct in event-based architectures).

Our choice to use these two rule types was motivated by the fact that these are the two most prevalent patterns of component behavior in the C2.FW/xADL 2.0 systems we were studying, and we believe that these patterns of interaction are also prevalent across a much wider class of event-based systems. We also found that it was possible to find events matching these rules by leveraging the SQL query capabilities of the event database.

Certainly, we cannot assert that these two rule types are completely sufficient to specify all possible component behaviors. *MostRecent* and *MatchingN* rules indeed have several limitations. For example, if a component that is supposed to respond to all events of a particular type but fails to do so, the *MatchingN* rule will correlate all messages after the failure incorrectly.

Also, there is always the chance of discrepancies between the specification of the rules and the system’s

actual behavior, arising from either incorrect specifications or behavior. This can lead to incorrect causality relationships or false positives; these situations will be discussed further in Section 4.

3.4. Visualizing application events

As noted above in Section 2.4, effective visualizations are an important part of understanding and debugging event-based systems. Minimally, our visualizations provide interactive access to the event log. Beyond this, we have built several coordinated visualizations in our prototype tools that allow the user to visualize the data created at the various points in our approach:

Event Capture: Users can visualize the structure of the system using the Archipelago xADL 2.0 editor, with or without trace connectors.

System Structure: Selected events, their potential causes, effects, and causal chains are all displayed on the visualization of the structure of the system.

Event Classification: Users can highlight events and architectural elements with visual distinctions, such as different colors, either manually or using classifiers. Users can also filter and sort the messages displayed based on classifiers, their source, destination, content, time, and process.

Event Causality: Users can explore causal chains interactively; when an event is selected, the list of possible cause and effect events is calculated, displayed to the user and graphically highlighted on the visualization of the structure of the system. The tools also aid in rule writing.

A screenshot of the tools can be seen in Figure 1.

4. Experience with the prototype

To evaluate our approach, we annotated components of two applications with rules and proceeded to examine the resulting message logs using MTAT. We wanted to verify whether our approach:

- Could be applied to an architecture without re-coding any components;
- Was useful in classifying event types;
- Supported our efforts of inferring component behavior;
- Could utilize incomplete rules to follow causal event chains through components; and
- Increased our understanding of an architecture with which we had no previous exposure.

4.1. Tracing Klax

KLAX is an interactive computer game that is highly asynchronous and interacts with the user in real-time. The configuration we examined is a single-process application with 16 components, built in the

C2 style [20]. It has an architecture specified in xADL 2.0. We instrumented Klax using the INSTRUMENTER and played a single game. The resulting trace contained approximately 40,000 messages.

We annotated KLAX with classifiers and rules twice, using different approaches. First, we extracted classifiers for events and rules for each component by looking only at the message log, treating all components as black boxes. Second, we used the source code for this task. We then compared both approaches.

When annotating KLAX with classifiers using the message trace, we used an incremental approach where we repeatedly wrote broad, coarse classifiers for events where we found commonalities. Then, we hid the events that matched that classifier, until we had classified all events coarsely. We then recursively applied this process to each group of events identified by a coarse classifier, making finer and finer distinctions among events. This approach allowed us to divide and conquer the process of classifying the events in the trace in a way that was mentally more manageable.

Next, we began inferring behavior rules for select components. We found that the easiest way to infer behavior was by starting with an emitted event as an effect and then searching for causes from events received recently. This approach was effective because, in KLAX, components receive a large number of events that they ignore since connectors do not filter them.

We found it advantageous that, in our approach, rules apply only to individual components; because of this we were able to develop a mental model of one component at a time. Causality chains emerged automatically with little or no intervention beyond single-component rule writing. We feel that this is the greatest strength of our approach.

This interactive classification and rule-writing process was assisted greatly by the fact that MTAT picks up new rules immediately and works with partial behavioral specifications. Thus, we did not need to restart our analysis every time we wrote a new rule (if we had, it would have nearly made analysis impractical). Further, MTAT tracks the current causal chain of events. This made it easy to go back to a previous event in the chain when we hit a dead end, or decided that the causal chain had become uninteresting.

After annotating KLAX “blindly” using only the message trace, we performed the annotation again, this time using the source code as an information source. Annotating the architecture from the source code took approximately 6 hours, which is slightly less time than it took to annotate KLAX from a trace, which took 8 hours. In both cases, the annotator was not an original KLAX developer; a developer would likely have been able to write these rules in a fraction of the time.

We compared the rules that we inferred by viewing the message trace to those obtained by looking at its source code. We found that the inferred rules were

occasionally incomplete due to message patterns that did not occur in our message trace. This is expected, as it is not possible to write rules for black-box components from a trace without total-coverage samples of their behaviors and reactions. However, even without the benefit of source code, we found that the resulting rules were surprisingly accurate. The rules obtained from the trace alone covered about 80% of the causality relationships revealed by the source code.

Interestingly, the rules determined from a trace of KLAX were more specific than those drawn from the source code. Rules written when viewing the source code typically encompassed multiple causal relationships because the commonalities of these relationships could be combined intelligently into one rule. Rules generated from the trace typically expressed only one relationship at a time because the commonalities between different rules were not as readily apparent.

4.2. Tracing AWACS Simulator

The AWACS Simulator is a distributed simulation of the software systems used on the US Air Force AWACS aircraft. We chose this system to test our approach's scalability because its size is typical of large, "real-world" systems. It consists of 126 components and 206 connectors. Like KLAX, the AWACS simulator has an architecture defined in xADL 2.0, which we automatically instrumented with the INSTRUMENTER. The instrumented architecture contained more than 400 trace connectors.

When working with the AWACS message trace of more than 50,000 messages, MTAT was not significantly slower when determining causal relationships than when tracing KLAX, and still performed responsively when interacting with the tool. This is because MTAT delegates much of the computationally intensive work of determining message causality to the underlying relational database (Oracle 8i in this case). As we stated earlier, we believe that leveraging the strong query capabilities of a modern relational database to implement classifier matching and causality is a key enabler of this approach.

4.3. Tracing MTAT

Because MTAT itself is written in the c2 style, we were able to use our tool to debug itself. This experience differed from the others because we were very familiar with the inner workings of MTAT.

In one case, when the user changed the color for a classifier, the MTAT GUI was not updating the colors on the various lists of events being displayed. We used the INSTRUMENTER to trace a run of MTAT exhibiting this behavior. By examining the trace, we could determine a number of things quickly: first, the GUI was sending out a `color_changed` event each time the user requested a color change. Second, by following

the potential effects of that event, we could see that the component in charge of this data was receiving the event. Third, we could see that it was sending out a `classifier_colors` event for each update, which specifies the current colors for each classifier. Eventually, we came to see that the component that was supposed to re-query the database using the new classifier colors, by sending out a query event, was not doing so. We determined that a typo was introduced when it was last updated causing it to ignore the event.

Other, more complex, situations occurred in which debugging was useful as well. However, this simple one demonstrates the usefulness of having such information, classifiers, and rules at your fingertips.

4.4. Lessons learned

In evaluating our approach, we found that the most obvious drawback—false positives—actually causes relatively few problems in practice. We identified the possible reasons for incorrect cause and effect lists, and how to determine actual causes and effects. More tool automation for this process could potentially reduce the time to ferret out correct causes or effects. Additionally, the incrementality of the approach means there is a balance between the number of false positives and how specific the rules are. It is up to the annotator to decide when the results are "good enough."

We have found that understanding a system with traces and causality relationships is easier and faster with the visualization tool than any sort of manual trace or code inspection. Its most basic capability is to provide a more manageable set of data, including data about causes and effects. For the applications we studied, a manual trace would have been infeasible simply due to the large amounts of messages and relationships.

5. Related Work

Many types of analysis, understanding, testing, and debugging tools have influenced our approach. These include architecture-based specification and analysis, state-based and specification-based analysis, notions of causality from the distributed systems and parallel computing communities, and debuggers for message passing systems.

Rapide [16] is an architecture description language and event pattern language that is compiled and executed as a simulation to find event sequences, causalities, and constraint violations. It is designed for prototyping system architectures. Rapide works with a causal event history and a set of events and relationships by creating a partial ordering of sets of events called *posets* [15]. Relationships can be defined using maps (aggregators) that list the input and output event patterns. Rapide's analysis of causality is based on complete component behavioral specifications, and is

decoupled from running systems (i.e. there are no tools for matching a running system to its specification).

Complex Event Processing (CEP) [14, 17], an extension to Rapide, assists in understanding of a system by organizing the activities of a system in an event abstraction hierarchy. CEP focuses on understanding a system by aggregating system-level events into higher-level events. Unlike our approach, it does not specifically take into account architectural topology. Furthermore, it is not heuristic; it expects complete component specifications and does not work well with incomplete information.

A number of approaches have used monitoring of distributed systems to evaluate whether a system meets its requirements. One approach is to determine when a system is being used in an environment for which it was not intended [9]. Another approach, FLEA (Formal Language for Expressing Assumptions) [5], takes as input expressions of requirements and assumptions, compiles them into runtime monitoring code, then generates notifications of any violations. These approaches differ from ours because they do not specifically account for architectural topology and cannot be dynamically applied using different expressions.

Other approaches to monitoring instrumented systems diagnose and manage network problems and performance, e.g. [11]. For program understanding and debugging, these tools tend to be too low level and lack techniques to identify the interesting messages from the uninteresting ones [17], although tools that do capture inter-component events are particularly useful for the event capture phase of our approach.

State-based analysis tools check whether a system meets its specifications, usually expressed in state-charts or a similar formalism. LTSA [3] (Labeled Transition System Analyzer) is a good example of this type of analysis. LTSA is a verification tool for concurrent systems that checks whether a system's property specification satisfies its actual behavior. In LTSA, the system and its properties are modeled as state machines. Analysis is based on compositional reachability, which searches for violations. A similar approach is seen in Balboa [6], which forms non-deterministic state-machine based event behavioral patterns from collected event data.

In general, state-based analysis techniques suffer from state-explosion problems, and usually require full formal models of system behaviors to be effective. Even with significant effort to reduce state space such as that seen in Nitpick [12] they are generally applicable only to extremely small systems. Additionally, many of these approaches focus on verification or analysis of early system artifacts rather than a running system, e.g. [1].

Some approaches produce finite-state models from implemented programs such as Java PathFinder 2 [2], JCAT [8], and Bandera [7], which can be analyzed

using model checkers. These tools differ from our approach because they do not address causal relationships between events dynamically created at runtime with other parts of a system, but rather analyze static descriptions of systems to ensure various properties.

The distributed systems and parallel computing communities have addressed issues of event causality in concurrent, distributed programs [18]. In these domains, processes are roughly equivalent to components in our approach. In this work, causality can help to establish a global system snapshot, which is important for certain kinds of analysis. Because our approach is focused on program understanding, global system snapshots are not relevant. Results from these domains concur that, in the general case, exact causality is difficult to determine. Determining potential causality, as our approach does, is feasible.

Finally, tracing-based approaches like ours have been used for debugging [4, 10, 13]. These combine message tracing with traditional functions of a debugger, such as breakpoints, stepping through execution, and examining system state. Some approaches add the ability to replay message sequences to achieve certain system states [19]. Most debugging-based approaches augment traditional program debuggers, which require source code for components [10]. Our approach to program understanding and debugging works with black-box components and allows a user to follow event traces that occurred at any time during a run.

6. Conclusion and Future work

This paper contributes a complete approach that aids in understanding, debugging, and visualizing the behaviors of event-based applications. The approach consists of four parts: 1) capturing events of real, implemented systems, 2) classifying the captured events into semantically meaningful categories, 3) dynamically determining causal relationships between events using rules specified with available knowledge or knowledge obtained as more is learned about the system, and 4) visualizing events with respect to the system structure, event classifiers, and causality rules. The approach supports partial and incomplete behavior specification as well as black-box components.

We evaluated our approach by implementing it for a particular event-based infrastructure and used it to analyze two different applications: KLAX and an AWACS simulator. We also applied it to itself. We believe our approach was useful in increasing the overall understanding of an architecture and its component interactions, but it did not always provide insight into the complete details of execution. It was, however, much easier to understand an application and its component interactions using our approach than any sort of code inspection, or manual trace, which would have been infeasible due to the large quantity of events and their relationships.

Our evaluation revealed that there are many benefits to using traces of real event-based systems. For example, if a component is not responding as expected, it is easy to verify whether it received a particular expected event, or if the event was emitted at all. The availability of causal chains makes it easier to solve a much wider range of problems. With chains, one can elide away irrelevant events and remain focused on a particular error. It also becomes easier to determine which components are actually using the services of other components, or how other components respond to a particular component's events.

Finally, our heuristic approach allows a user to begin with little or no understanding of a system. This is especially useful when trying to understand a new system, or a different part of the system with which a component did not previously interact.

The beneficial properties of event-based architectures, combined with increasing support from practitioners and researchers, means that it is likely that more and more event-based systems will be created. However, lack of end-to-end development and maintenance support for such architectures could hinder adoption and raise the costs of building event-based systems. Our approach contributes a usable, viable means to understanding event-based systems, but it also exposes several important issues in event-based development that we plan to investigate. These include using event tracing as a basis for testing and debugging, the role of heuristic techniques to find "good enough" answers to development problems, and finding novel ways to deal with the deluge of events that occur in even moderate-sized event-based systems.

Our evaluations so far show that our approach shows promise for understanding the systems we looked at. We believe that our approach is generalizable to a large class of event-based systems including those that are dynamic, use more complex event patterns, and that are highly distributed. Future work needs to focus on applying our approach to these types of event-based systems to evaluate its feasibility and effectiveness.

7. References

[1] Atlee, J.M. and Gannon, J.D. State-Based Model Checking of Event-Driven System Requirements. *IEEE Trans. on Software Engineering*. 19(1), p. 24-40, January, 1993.
[2] Brat, G., Havelund, K., et al. Java PathFinder - A second generation of a Java model checker. In *Proc. of the Workshop on Advances in Verification*. Chicago, July 20, 2000.
[3] Cheung, S.C. and Kramer, J. Checking subsystem safety properties in compositional reachability analysis. In *Proceedings of the 18th Intl. Conference on Software engineering*. p. 144-154, Berlin, Germany, March 25-29, 1996.
[4] Cláudio, A.P., Carmo, M.B., et al. Monitoring and Debugging Message Passing Applications with MPVisualizer. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*. Rhodes, Greece, January 19-21,

2000.

[5] Cohen, D., Feather, M.S., et al. Automatic Monitoring of Software Requirements. In *Proceedings of the 19th International Conference on Software Engineering*. p. 602-603, Boston, MA, May, 1997.
[6] Cook, J.E. *Process Discovery and Validation through Event-Data Analysis*. Ph. D. Thesis. University of Colorado, 1996.
[7] Corbett, J.C., Dwyer, M.B., et al. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*. p. 439-448, Limerick, Ireland, June 4-11, 2000.
[8] Demartini, C., Iosif, R., et al. Modeling and Validation of Java Multi-threading applications using Spin. In *Proceedings of the 4th Workshop on Automata Theoretic Verification with the Spin Model Checker*. p. 5-19, Paris, France, November, 1998.
[9] Fickas, S. and Feather, M.S. Requirements Monitoring in Distributed Environments. In *Proceedings of the 2nd Intl Workshop on Services in Distributed and Networked Environments*. p. 93-100, Whistler, BC, Canada, June, 1995.
[10] Frumkin, M., Hood, R., et al. Trace-driven debugging of message passing programs. In *Proc. of the First Merged International Symposium on Parallel and Distributed Processing*. p. 753-762, Orlando, FL, Mar 30-Apr 3, 1998.
[11] IBM. *Tivoli Monitoring - Product Overview*. <http://www-306.ibm.com/software/tivoli/products/monitor/>, IBM, Website, 2004.
[12] Jackson, D. and Damon, C.A. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*. 22(7), p. 484-495, 1996.
[13] Lencevicius, R., Ran, A., et al. Third eye — specification-based analysis of software execution traces. In *Proceedings of the 22nd international conference on Software engineering*. p. 772, Limerick, Ireland, 2000.
[14] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 400 pgs., Addison-Wesley, 2002.
[15] Luckham, D.C., Vera, J., et al. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Systems and Software*. 21(3), p. 253-265, June, 1993.
[16] Luckham, D.C. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. In *Proceedings of the DIMACS Partial Order Methods Workshop IV*. Princeton University, July, 1996.
[17] Luckham, D.C. and Frasca, B. *Complex Event Processing in Distributed Systems*. Stanford University, Report CSL-TR-98-754, March, 1998.
[18] Schwarz, R. and Mattern, F. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*. 7(3), p. 149-174, 1994.
[19] Spiteri, M. *An Architecture for the Notification, Storage, and Retrieval of Events*. Ph.D. Thesis. Darwin College, University of Cambridge, 2000.
[20] Taylor, R.N., Medvidovic, N., et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. 22(6), p. 390-406, June, 1996.