ISR
Institute for Software Research
University of California, Irvine

An Approach for Tracing and Understanding Asynchronous Systems

Scott A. Hendrickson
Univ. of California, Irvine
shendric@ics.uci.edu

Richard N. Taylor
Univ. of California, Irvine
taylor@ics.uci.edu

Eric M. Dashofy
Univ. of California, Irvine
edashofy@ics.uci.edu

Santiago Li
Univ. of California, Irvine
lis@uci.edu

Adrita Bhor
Univ. of California, Irvine
abhor@ics.uci.edu

Nghi Nguyen
Univ. of California, Irvine
nghin@uci.edu

December 2002

ISR Technical Report # UCI-ISR-02-7

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2002		**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An Approach for Tracing and Understanding Asynchronous Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**12**	

An Approach for Tracing and Understanding Asynchronous Systems

Scott A. Hendrickson, Eric M. Dashofy, Adrita Bhor, Richard N. Taylor
Institute for Software Research
University of California, Irvine
{shendric, edashofy, abhor, taylor}@ics.uci.edu

Santiago Li, Nghi Nguyen
Information & Computer Science
University of California, Irvine
{lis, nghin}@uci.edu

ISR Technical Report # UCI-ISR-02-7

December 2002

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 set of requirements for an aid to assist in exploring the behavior of such applications, with the aim of assisting in the development and understanding of such applications. A prototype tool is presented, indicating viable approaches to meeting requirements. Experience with the tool reinforces some of the requirements and indicates others.

An Approach for Tracing and Understanding Asynchronous Systems

Scott A. Hendrickson, Eric M. Dashofy, Adrita Bhor, Richard N. Taylor, Santiago Li, and Nghi Nguyen

Institute for Software Research, University of California, Irvine
Irvine, CA 92697-3425
+1 949 824 4101
{shendric, edashofy, abhor, taylor}@ics.uci.edu, {lis, nghin}@uci.edu

ISR Technical Report # UCI-ISR-02-7

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 set of requirements for an aid to assist in exploring the behavior of such applications, with the aim of assisting in the development and understanding of such applications. A prototype tool is presented, indicating viable approaches to meeting requirements. Experience with the tool reinforces some of the requirements and indicates others.

1. Introduction

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

Understanding an event-based application without support tools and methods is a difficult task, due to the large number of events flowing through an architecture, the complex, asynchronous interactions among components, and the lack of explicit mechanisms within the application for understanding causal relationships between individual events. Since the interactions in an event-based system are so different from those of a tightly-coupled, synchronous system (like most object-oriented systems being built today) tools that work well on them, such as traditional program debuggers, usually work poorly on event-based applications. Additionally, components in an event-based system may be created and maintained by an outside agency, such as a third-party software developer. In this case, source code or specifications for some components may be unavailable. Thus, it is useful to have techniques for understanding and testing event-based architectures that do not rely on the presence of source code or formal behavioral specifications for a component.

1.1. Objectives

This context suggests a set of broad challenges for tool support for aiding understanding of the behavior of an event-based system.

- How can an event-based architecture be instrumented such that events can be gathered for viewing and analysis?
- How can causal relationships between messages in an event-based architecture be determined?
- How can messages be organized and visualized to cultivate a higher understanding of the system?

Our experience with building and evolving event-based systems, as well as the experience of others, led us to refine these general objectives to the following goals:

Message Capture:

- Message capture should be the primary source of data about the system. Approaches must be able to deal with components without available source code or formal behavioral specification, due to constraints of the environment.

- Event acquisition should minimally disturb application characteristics. Some effect on application performance must be expected, but semantic changes should be avoided.

Message Relationships and Causality:

- Causal relationships among messages must be determined without access to or modification of component source code, again, due to environmental constraints.
- Determination of causality relationships need not always be accurate, but any inaccuracies in the approach to identifying causality should be accompanied by methods for a human to identify and weed out inaccurate results.
- Any specifications needed to identify causality (above and beyond topological architecture descriptions and system traces) should be usable and applicable to complex, off-the-shelf components.

Presentation:

- Though the analysis is grounded in data from the implementation, results should be correlated and presented to the analyst in terms of events between components (i.e. at the architectural level).
- Visualization tools and/or techniques should be provided that present the data in a way that does not overwhelm a user.
- Visualization tools that provide multiple views of the data, and multiple methods of filtering those views, are preferable.

Approaches that work in environments with distributed or dynamic event-based systems are preferable. Additional goals we have identified but not fully explored in our approach focus on the use of tracing and causality for objectives beyond simple program understanding. We believe that good approaches can help support test case generation, system debugging at the architecture level, and possibly formal or semi-formal analysis.

1.2. Overview of Approach

We developed an approach that meets the basic goals above and evaluated its feasibility through creation and use of a prototype tool. The tool examines the architectural description of an event-based system, specified in xADL 2.0 [5], and modifies it by interspersing "trace connectors" into the description. Trace connectors log messages that are exchanged between components. Because the architectural description is used to instantiate and configure the architecture, not to govern a component's internal structure, no component's source code is changed. The system architect also annotates the architecture description with rules that describe, for each component, considered as a black box, expected causality relationships among message types. Finally, a graphical tool allows the user to interact with the message log and explore message causality chains, using the specified rules as guides.

Rules are interpreted heuristically, meaning that they may falsely indicate that a causality relationship exists between two messages when in fact there is none, but empirical use of our approach has revealed that this inaccuracy has minimal practical impact. Finally, a visualization tool is used to display and explore the causal relationships resulting from the trace and the applied rules.

We applied our approach to two different event-based applications: KLAX and ArchStudio 3. KLAX is an interactive computer game specifically built to be highly asynchronous and interact with the user in real-time. ArchStudio 3 is a larger architecture with some very complex components and a mixture of synchronous and asynchronous interactions. A subset of the authors applied the approach and tools to these architectures without first-hand knowledge of how they were developed, although we chose them partially because the original developers were available to answer questions and help us to verify our results.

While our approach could not give us a complete and authoritative understanding of how the event-based systems worked, we were nonetheless surprised at how much useful information could be obtained from this approach. We were able to step through whole architectures following select messages of interest, and found that our approach is a significant improvement over manual tracing, and augments other techniques for program understanding, like reading code, well.

2. Background

Our work has been influenced by other types of analysis, understanding, testing, and debugging tools for both event-based and tightly coupled systems. We review selected approaches here.

Rapide [11] is an architecture description language or more specifically, an event pattern language, which 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 order of sets of events called *posets*. Relationships can be defined using maps (aggregators), which list the input event patterns and output event patterns. In terms of the goals outlined above, Rapide's analysis of causality is based on complete behavioral specifications of components, and is

decoupled from running systems (i.e. there are no tools for matching a running system to its specification). This limits its suitability for use in the context of implemented, possibly COTS-based systems.

Complex Event Processing (CEP) [10][9], an extension to Rapide, shares a similar goal and assists in understanding of a system by organizing the activities of a system in an event abstraction hierarchy. CEP also introduces the concept of Event Processing Agents (EPAs) and Event Processing Networks (EPNs). An EPA is a simple object that consists of some component state, and rules with a "trigger" and a "body". The trigger indicates the event sequence that will fire the body, which consists of actions. Actions may modify the state of the EPA, or emit output events. Like Rapide, CEP operates mostly on simulations and system specifications, rather than on implemented systems.

Another popular approach towards analysis and verification is state-based analysis. LTSA [2] (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. State-based process discovery and validation can be seen in Balboa [4], which is based on formation of non-deterministic state-machine event behavioral patterns from the collected event data. State-based analysis can suffer from state-explosion problems, and usually require full formal models of system behaviors to be effective. Furthermore, few state-based analysis approaches provide the user with an interactive visualization of the information, allowing them to make higher-level inferences.

Tracing-based approaches like ours have been used for debugging [3][6][8] or performance analysis in distributed systems [12] and in parallel programs [7]. They have rarely been used to facilitate understanding of event-based architectures. When debugging, instrumentation and event collection is done at the source code level and analysis is usually based on traditional static analysis techniques like dataflow analysis with data and data relationships and/or control-flow analysis with call graphs, control flow graphs or program dependence graphs (PDGs). These techniques are usually rooted in source code analysis, and do not apply well to traces of black-box components or higher-level events like messages in event-based architectures.

3. Approach

As summarized earlier, we developed a three-part tool-based approach:

1. One tool modifies the architectural description of a system by inserting "trace connectors" into the description. The system is then executed with these trace connectors in place, which log all messages sent between components and connectors in the system.
2. The system architect annotates the architecture description with rules that describe, for each component, expected causality relationships among message types based on the known behavior of the component.
3. A graphical tool allows the user to interact with the message log and explore message causality chains, using the specified rules as guides.

We applied our approach to two systems built in the C2 style [13], a style that is representative of event-based architectural styles in general, to see if it was useful or effective in giving us a better understanding of systems with which we had limited previous exposure. The results of our evaluation are detailed in Section 4.

3.1. Gathering architecture events

A message trace is used as the primary basis for understanding the communication among components in an architecture. This trace contains a log of all the messages sent in the architecture during an execution of the system. In our approach, real, implemented systems are instrumented, rather than relying on a specification that may or may not match the implemented system. This is contrasted with approaches like Rapide's [11] and CEP's [9], which rely solely on a specification for information about an architecture's behavior.

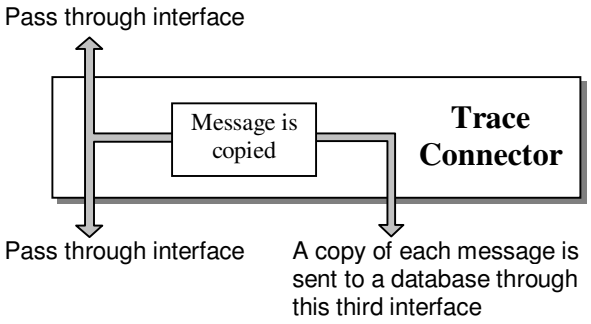

Figure 1. Trace connector

The obvious place to instrument an architecture to log all messages is in the underlying middleware or architecture framework, since all messages are handled by that framework. However, this would bind our approach to a specific framework or middleware. Instead, we developed an approach that works by modifying the architecture of the system itself, but, we would argue, in a rather benign manner.

We collect message traces by instrumenting an architecture with *trace connectors* (see Figure 1). Trace connectors intercept all messages passing through them, make a copy of each message, and send this copy to a distinguished component that logs each message to a relational database. The original messages are passed on unmodified. These trace connectors are first-class connectors, and are not part of any component in the architecture. Figure 2 shows an architecture before modification as well as the architecture after inserting trace connectors, which would be instantiated to gather the application's messages.

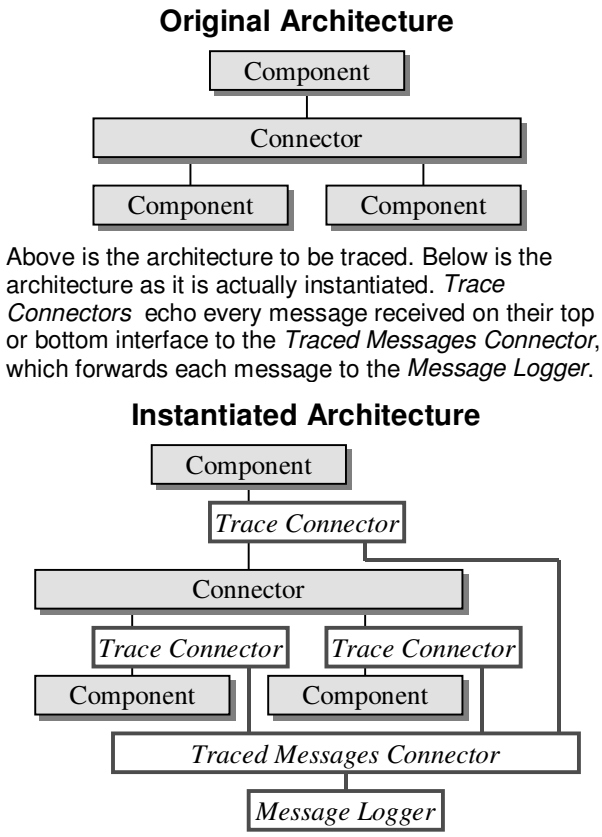

Figure 2. Gathering application events

We developed a prototype tool that examines a xADL 2.0 description of a system's structure and inserts trace connectors into that description. The system's structure is modified so that each link in the original architecture is split into two links, with a trace connector in between. Because the infrastructure we used for our prototype instantiates architectures directly from their descriptions, no recoding is needed for instrumentation. Consequently, components in the application remain unaware of any architectural changes or the presence of the trace-connectors once the architecture is modified.

Instrumentation, of course, results in an impact on the overall speed of the application, which we found to be dependent on the message traffic in the system (see Section 4 for details). However, this is inevitable in single processor systems since message logging necessarily produces extra load on the system.

3.2. **Determining causal relationships**

A critical part of understanding an architecture is understanding the causal relationships between messages. Our approach treats all components as black boxes (which some may truly be, due to lack of available source code). Thus, all that can be observed about in architecture are the messages sent between components and connectors. This is a hard environmental constraint, and distinguishes our approach from those like Rapide CEP, and LTSA, which need formal models to operate correctly.

Without explicit, complete models of the internal functions of components, which would likely be impossible or impractical to create, it is not possible to know with certainty the causal relationships between messages. That is, if a component receives message 'A' on one interface, and later sends out a 'B' on another interface, it may be that the receipt of 'A' caused the component to emit 'B,' but it is impossible to know for sure. However, we have found that it is possible to develop a simpler model of component behavior that can indicate that the receipt of 'A' *probably* caused the emission of 'B' with a high degree of accuracy.

Of course, an ideal situation would be if the component itself tagged each emitted message with a list of 'caused-by' messages. However, this would require cooperation from the component developer or access to the component's source code, which cannot be assumed in this environment. Rather than resort to modeling a component's behavior with a finite state machine or some other complete modeling formalism (which is impractical for many reasons, chief among them state explosion) as some approaches require, we have developed a simple language of *rules* that are used to specify causal relationships between messages. Rules are a property of a component, and are specified by the software architect (or possibly the original component developer). Rules are *not* complete specifications of a component's behavior; rather, they describe, at varying levels of abstraction, how a component reacts to messages.

Each rule defines a causal relationship by specifying a set of *causes* and a set of *effects*. Causes and effects are specified as sets of message characteristics, rather than specific message types or contents. For example, a cause might be described informally as "any message with the name 'A'" rather than describing the entire contents of the message.

In general, event-based applications may use any sort of structure for messages, so long as they obey the general rules of the architectural style (messages may not contain

pointers or control objects, etc.) This means that messages may be amorphous (a 'bag of bits') or highly structured (like an XML document). Knowledge of the message structure is required to express rules in our approach. For instance, to express a rule that matches messages by name, it must be understood that messages have names. For our prototype tools and rule language, we assume that messages consist of a name string and a set of name-value pair properties. Property names are character strings, and property values are arbitrary objects. The property set for a message may be empty. We chose this format because it is the one used by most applications built in the C2 style, including those that were the target of our evaluation.

For each cause and effect, in addition to message characteristics, a number of required occurrences is specified, as well as the interface on which a message would be received or emitted. For purposes of discussion, we will specify message in this paper as follows:

message *name*{
property_name1 = *property_value1*;
property_name2 = *property_value2*;
…
}

For example, if a component in a game application requests the scores of the home and away teams it might emit a message:

message *request_team_scores*{ }

A component that knows the scores, upon receiving this message, might emit two responses:

message *current_team_score*{
team = *"home_team";*
score = *21;*
}

message *current_team_score*{
team = *"away_team";*
score = *17;*
}

When we define our rules, we refer to a name and/or a set of property name/value pairs that an event must have in order to match that rule. For example, if we wanted to define a rule that applies to all events requesting scores, we would specify that the event has to have the name *request_team_scores*.

We define two types of rules: *MatchingN*, and *MostRecent*. *MatchingN* indicates that a component will *always* send the complete set of effect messages when it receives a complete set of cause messages. Therefore, the n^{th} set of cause messages will always be associated with the n^{th} set of effect messages. Asking for the effects of the fourth *request_team_scores* messages above would yield the seventh and eighth *current_team_score* messages. Likewise, asking for the causes of the seventh or the eighth *current_team_score* message would yield only the fourth *request_team_scores* message. This rule type is especially applicable to components that queue up requests or that broadcast messages such as a C2 connector.

The rule that identifies the causal relationship for the component that keeps score in our example would look something like Table 1.

Table 1. Request score rule

Request Score Rule	
Rule Type**:**	*MatchingN*
Set of Causes**:**	
1.	message *request_team_scores*{ }
Set of Effects**:**	
1.	message current_team_score{ *team = "home team"* }
2.	message current_team_score{ *team = "away team"* }

The other rule type, *MostRecent*, indicates that a component will respond to a set of events immediately. The component that knows the score in a game may respond with a *game_over* message after receiving an *increase_team_score* message once a goal score is reached by either team. Applying a *MostRecent* rule to such a component would only associate the *game_over* message with the most recent *increase_team_score* message (which would be the last one received before the game ended). *MostRecent* rules are applicable towards components that respond immediately to messages they receive.

MostRecent and *MatchingN* rules are subject to the following limitations:

- If a component that is supposed to respond to all events of a particular type by emitting a message fails to do so for one of those events, the *MatchingN* rule will correlate all messages after the failure incorrectly,
- If a component emits responses more slowly than it receives requests then *MostRecent* may incorrectly identify relationships between the requests and responses, and
- There may, of course, be discrepancies between the specification of the rules and the system's behavior, arising from either incorrect specifications or behavior.

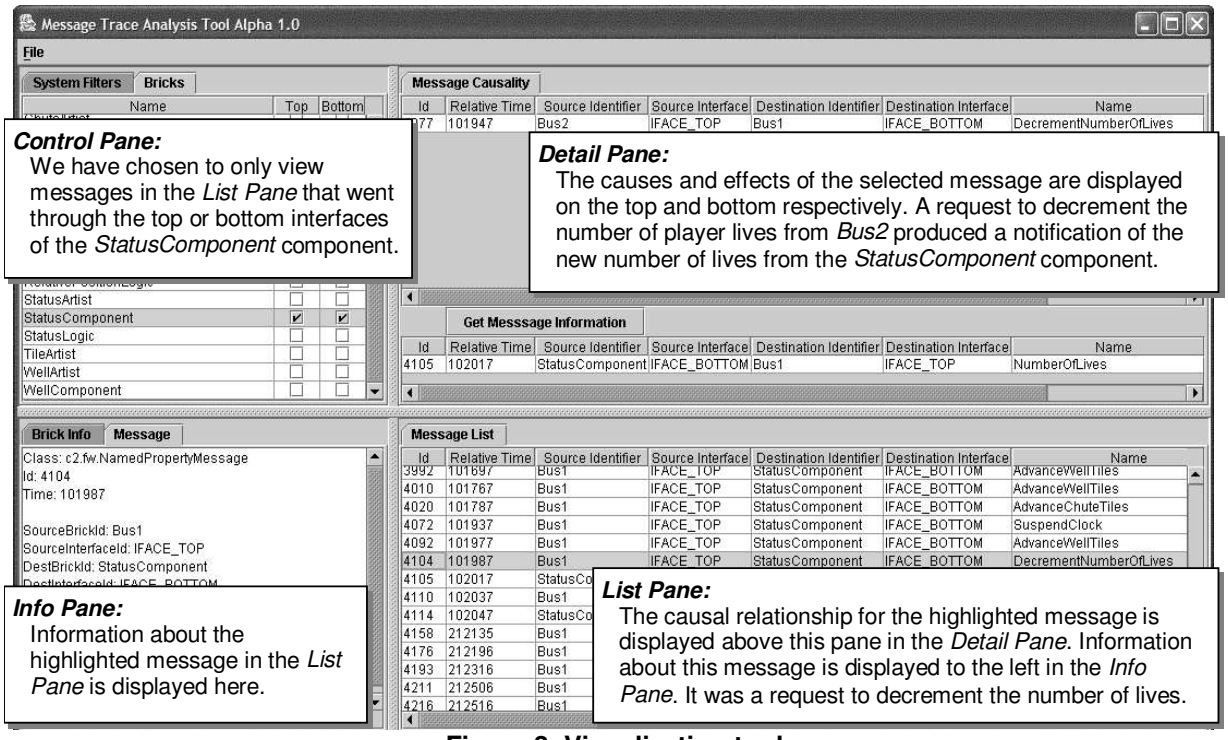

Figure 3. Visualization tool

An interesting issue arises with these rules because of the asynchronous nature of event-based systems. There is no limitation on the time a component can take to process a message. Thus, in our example above, if the component that knows the team scores decides to emit the team scores of its own accord (perhaps it does so on a periodic basis), those score messages may incorrectly be reported as being caused by a *request_team_scores* message that happened earlier in the system execution. Usually, this kind of false positive can be identified by an abnormal amount of time passing between the cause and effect messages.

When multiple rules apply to the same cause message(s), the results are combined. It might be that the rules complement each other and the combined results are valid. This might result if the rule in Table 1 were split into two separate rules, one for the "*home team*" effect message and one for the "*away team*" message. It is also possible that the combined results contain effects that are mutually exclusive (i.e. one of two rules applies to a particular situation, but not both at the same time). In this case, the user must determine which effect messages are actually valid by examining them.

It is best to have rules that err on the side of including a false positive rather than rules that are too restrictive and may exclude a message that is the actual cause or effect. It is easier to find the true cause from a small list of potential causes than it is to find the true cause from the whole message log. Without an appropriate rule or a rule that is too restrictive, a causal relationship will not be presented. However, this aspect of the approach can be used advantageously by the rule writer since rules can be written that deliberately omit 'uninteresting' causal relationships.

We believe that *MatchingN* and *MostRecent* rules capture many, if not most, of the types of causal relationships that occur in event-based systems. An additional virtue of these rules is that they can be used to programmatically find both causes and effects for a given message. While there are bound to be some erroneous results, we found them to be, in practice, minor.

Our approach trades off rule-writing effort for accuracy. Less complete specifications (or those with more general rules) will produce larger sets of possible cause and effect messages. More complete specifications will tend to narrow these lists.

To test the viability of the rules we proposed above, we extended xADL 2.0 with syntactic constructs that allow a rule-writer to specify rules for components and connectors. We then annotated several event-based applications' architecture descriptions with appropriate rules; our experience doing this will be described in Section 4.

3.3. **Visualizing application events**

A tool to visualize the traced events should:

- Provide access to the complete and correct message log generated by the application;

- Provide the ability to filter the log to view messages of particular interest both through the use of custom filters and through the ability to examine a single component/interface combination;
- Provide the ability to "zoom in" on a particular message or component to see its parameters, properties and property values;
- Clearly display a list of possible causes and effects of a given event by implementing rule-based searches on the execution trace allowing the user to follow a chain of causes or effects; and
- Provide a means to verify whether a reported cause or effect is accurate.

We believe that the user's ability to interact with the trace and causality data in a system is critical. Event-based systems generate too much information to be viewed all at once. Contrast this with Rapide's event-causality graphs, which can be so complicated as to be unreadable. An interesting future direction suggested by our work is to explore how different visualization techniques and views can assist the user in gaining a better understanding of the system.

We developed a prototype graphical tool that allows the user to interact with the message log and explore message causality chains. The tool supports applications written in the C2 style (and is itself written in the C2 style) but could be easily modified to other event-based topologies.

The main screen of the display is segmented into four sections (See Figure 3):

- The *Control Pane* allows a user to apply and modify custom filters that aid a user in finding a particular message or set of messages of interest independent of causality relationships, view components of the system, and select components of interest for analysis.
- The *List Pane* displays messages in the message log.
- The *Info Pane* displays "zoomed in" information about a message or component as well as options for a filter.
- The *Detail Pane* displays lists of potential causes and effects of any given message.

Our tool allows a user to view the message log in its entirety. However, users may limit the display to messages of a particular component/interface combination and/or applying custom filters which exclude messages outside a given time frame. Detailed information for each message or component is displayed in the info pane when the message or component is selected in the GUI.

When a user selects a message in the list pane, the detail pane displays two lists of messages. The top list is the list of potential causes of the selected message and the bottom list is a list of its potential effects. A user may double click on a message in one of these lists to change the selected message, thus updating the detail pane to show the newly selected message's lists of causes and effects. In this way, a user may "walk through" a message causality chain.

Due to the heuristic nature of the approach, and possibly inaccurate rules, false causes and effects may be displayed, or true causes and effects may not be displayed. There are a few hints that generally indicate that one of these is the case:

- The number of potential causes or effects listed is not equal to the number expected, or
- The span of time between a message and a potential cause and effect is surprisingly large.

There are four possible reasons for an inaccurate list of causes and effects:

- The limitations of the *MostRecent* and *MatchingN* rule prevent correct assessment,
- A rule is incorrect,
- The expected event did not occur in the application, or
- The application has a bug that prevented the appropriate event from occurring.

When a list is incorrect, it is necessary to determine which, and if any, of the effects or causes are correct, which are incorrect, and whether true causes and effects were reported at all. It is always possible to accomplish this by examining the unfiltered message log since it is complete. If the event did not occur and it should have, then this may indicate a bug in the component. If the event did occur, but the effects and causes are not listed, then there is likely a defect in the rule specification. In practice, we have found that the most common inaccuracy is false positives. To ferret these out, it is usually adequate to view the details of each message and the rules of appropriate components to determine which messages are the false positives and which are not.

4. Experience with the prototype

To test our approach, we annotated components of two applications and proceeded to analyze the resulting message logs. We wanted to verify whether our approach allowed us to:

- Apply our approach to an architecture without recoding any components;
- Follow causal chains through components; and
- Understand an architecture with which we had no previous exposure.

Table 2. KLAX 'WellComponent' rule

KLAX 'WellComponent' Rule	
Rule Type**:**	*MostRecent*
Set of Causes**:**	
1.	message *minor_tick*{ }
Set of Effects**:**	
1.	message *advance_tile*{ }

4.1. **Tracing KLAX**

KLAX is an interactive computer game that is highly asynchronous and interacts with the user in real-time. We decided to trace KLAX because we had no previous knowledge about how KLAX works or how it was developed. It is a moderately complex single-process application with 16 components in all.

Because we had the source code for KLAX available, one member of our team was tasked with annotating it by reading through the code. Annotating the architecture took approximately 6 hours, which was longer than we anticipated. However, this only has to be done once for an architecture, so it was a reasonable expenditure of time. Furthermore, our annotator was not an original KLAX developer; the architect who originally designed KLAX might benefit from previous knowledge and accomplish the same task in a fraction of the time. The annotator used xADL 2.0 tools to write the rules into the existing KLAX specification. We instrumented the KLAX description with our automated tool; the resulting architecture was not significantly slower than the original architecture. This is likely due to the fact that the game is synchronized to messages emitted by a clock component, which causes the application to have a significant amount of idle time in which messages may be logged.

The tool captured approximately 7000 messages after 40 seconds of execution. We began by stepping through the causal chain rooted at the very first application message logged in our database and found that a design diagram of the architecture was necessary to avoid getting lost within the architecture. We were able to progress down through various chains of effects and causes.

On occasion we would come across a causal chain that would stop unexpectedly. After investigation, we found that in these cases there were typographical errors in the rule specification. Once the rule specification was corrected, the causal chain was also fixed. We also encountered situations where false effects would be listed, but these were easy to weed out by examining the relative times of the messages as well as their contents.

Additionally, the limitation to the *MostRecent* rule caused some confusion. A component in KLAX named the 'WellComponent' implemented the rule specified in Table 2.

We know from the way the game works that when the game is started, the 'WellComponent' does not emit any *advance_tile* events until the game has been played for a few seconds. The game clock component emits *minor_tick* events many times every second. This means that the very first *minor_tick* event couldn't possibly cause an *advance_tile* event, yet this is exactly what our tool reported. It turns out that because the 'WellComponent' does *eventually* follow this rule, there is a valid *advance_tile* event in the message log that occurs after the first *minor_tick* event. Consequently, the rule reports the *advance_tile* event as an effect of the first *minor_tick* event when it was really the effect of a *minor_tick* event that happened later in the application execution. We were able to verify this by looking at the message log as well as observing that the listed cause of the first *advance_tile* event was not the first *minor_tick* event.

Minor setbacks aside, we felt that our understanding of KLAX was increased using our approach. We gained a general understanding of the system and its expected behavior.

4.2. **Tracing ArchStudio 3**

ArchStudio 3 [1] is an architecture-based development environment created at UCI. It consists of components for manipulating and evaluating architecture descriptions and their implementations. ArchStudio is a more complex application than KLAX, containing approximately 20 components. In terms of code size, ArchStudio 3 is approximately three times larger than KLAX in terms of lines of code, indicative of the complexity of some of its components.

We annotated 17 of the ArchStudio components with rules in approximately 5 hours. The trace of ArchStudio was successful, but the impact on tracing speed was more prevalent with a 50% performance hit. We believe that the speed is a more significant issue for ArchStudio than for KLAX because ArchStudio produces a higher volume of messages to be logged: approximately 18000 messages in 15 seconds.

After applying our approach and conferring with the original developers of ArchStudio 3, we found that the trace of the execution gave us a general understanding of the system's expected behavior. The trace confirmed the behavior of certain components. For example, the trace showed InvokableStateMessage messages sent during the beginning of the execution, which corresponds to the expected initialization behavior of invokable GUI components (e.g. ArchEdit). Some of the messages were easily associated with the user's actions during execution. By examining the message contents of messages originating from xArchADT, the data repository component, we were able to note when and how open

architecture files were accessed. Furthermore, the messages were traceable back to the ArchEdit tool, which showed that the user was editing an architecture file with ArchEdit. Messages with warnings concerning the architecture file indicated that the critic framework was active. With closer examination, the exact critic reporting the issue(s) could be identified.

The tool was ineffective at times when the rules were specified incorrectly. The annotations for the most part corresponded to the observed behavior, but a few inaccuracies were discovered in the message parameters of the rules when examining the trace. These inaccuracies were fixed, our tool restarted, and the messages were reanalyzed for causality. Consequently, the tool displayed the expected results according to the original architect.

In ArchStudio 3, some messages did not have enough information to identify the purpose of the message when viewed in our visualization tool. As discussed earlier, in a message, the property values of a message may contain arbitrary—that is, binary—data. The tool will identify the data as binary instead of displaying the actual data, but this is insufficient for a person to determine whether the potential causes and effects for a message are correct or not. The tool is currently limited to displaying strings and numeric values. The messages for KLAX contained visualizable information (strings and numbers) so this was not a problem for KLAX, but the messages for ArchStudio sometimes contained other types of objects. This problem indicates how important the human user's participation is in the process. Our visualization tool could be easily enhanced to support different types of data if necessary.

4.3. **Lessons learned**

In the evaluation of our approach, we learned where it is useful, as well as the incidence and severity of several drawbacks. Most notably, the accuracy of causality relationships was questionable at times due to the limitations of the rules concerning the *MostRecent* and *MatchingN* interpretations discussed earlier, and the more accidental limitations on viewing binary message contents. Since the tool heuristically determines causality with rules, the identified causality of a message may not be correct.

We found that the ease of understanding a system through traces varied from application to application. KLAX messages contained clear state information, but ArchEdit often contained message content that only had meaning at runtime, such as numbers that represent information relevant to a particular component or pair of components during runtime, but not explicit state data.

The tool is helpful for understanding the general behavior of a system, but the details of the execution may not be clear at times. However, 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, with some data about causes and effects. For the applications we studied, which are moderately complex, a manual trace would have been infeasible simply due to the large amounts of messages and relationships involved.

5. Future work

The specification language that we used for rules worked well with the applications we examined, however, we did find that it could be adapted to increase its expressability. The ability to specify conditional properties, such as *name* > *value*, along with the ability to specify property values other than strings, such as binary data, would be beneficial.

One of the most common reasons we found for erroneous results was an incorrectly specified rule. Generally, the error was the result of a typo such as specifying the message name as *advanced_tile* rather than the correctly named *advance_tile*. It would be very useful to determine if all rules specified in a system applied to at least one instance of the observed messages of a system. This would provide an easy way for the architect to avoid minor specification errors.

Another useful feature would be the ability to analyze an application while it is running. In this way, a user may trigger an event in the program, see what messages were produced, then trigger another event. This would aid the user in associating actions performed by the application to messages processed within the architecture. The ability to examine a dynamic application, whose architecture changes at runtime, would also be useful.

A graphical layout of the architecture during visualization would greatly help the user to keep track of where a message is in the greater context of the architecture. Adding appropriate highlights to the architecture to indicate the components involved in a message's causality would also aid in understanding. Also useful would be a history of the causality chain being explored so that a user may jump back to previous causal links for further investigation of a particular causal relationship.

In the long-term, we believe that message tracing and causality relationships have the potential to be valuable in other parts of the software development process. Already, we have seen how causality relationships can indicate bugs or incorrect rule specifications in an architecture, indicating their usefulness in debugging and possibly requirements specification. We believe that message traces and rules can also be useful in aspects of testing, such as test-case generation.

6. Conclusions

This paper contributes a set of goals and future directions for using event traces as a basis for aiding developers in the creation and maintenance of event-based systems, as well as an approach that demonstrates the validity and usefulness of some of these goals. The approach includes tools and techniques to gather a complete trace of events of an event-based application independently of the specific framework used for that application. A means of determining causal relationships between the gathered events of an application by using a heuristic approach has been implemented and found to be useful. The rule set used to specify these causal relationships is simple and usable, not fully formal, and applies to systems where component source may not always be available. A visualization tool is presented that displays message causality and provides a means of verifying whether or not reported causes and effects are accurate given a set of causality rules and a message log of a running application.

We demonstrated the effectiveness of our approach by using it to analyze two different applications: KLAX and ArchStudio 3. We found that the ease of understanding a system through traces was increased when messages contained clear state information, but was more difficult when contents had meaning within contexts created at runtime. The approach was useful in increasing the overall understanding of an architecture but did not always provide insight into the complete details of execution. It was, however, much easier to understand an application using the analysis tool than by any sort of manual trace or code inspection. A manual trace of the applications we studied would have been infeasible due to the large amounts of messages and relationships involved.

The beneficial properties of event-based architectures and 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 approach to understanding complex 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.

7. Acknowledgements

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

8. References

[1] ArchStudio 3. URL: http://www.isr.uci.edu/projects/archstudio/

[2] Cheung, S.C. and Kramer, J. "Checking Subsystem Safety Properties in Compositional Reachability Analysis", *Proc. of the 18th Int'l Conference on Software Engineering*, Berlin, Germany, March 1996, pp. 144-154.

[3] Claudio, A.P.; Cunha, J.D.; Carmo, M.B; "Monitoring and debugging message passing applications with MPVisualizer," *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing*, 2000.

[4] Cook, J. E., "Process Discovery and Validation Through Event Data Analysis", *Ph.D. Thesis, Department of Computer Science*, University of Colorado, Boulder, Dec. 1996.

[5] Dashofy, E.M., van der Hoek, A. and Taylor, R.N., "A Highly-Extensible, XML-Based Architecture Description Language", *Proceedings of the Working IEEE/IFIP Conference on Software Architectures*, Amsterdam, Netherlands, 2001

[6] Frumkin, M.; Hood, R.; Lopez, L. "Trace-driven debugging of message passing programs", *Proceedings of the First Merged International Symposium on Parallel and Distributed Processing*, 1998.

[7] Kraemer, E.; Stasko, J.T., "Issues in visualization for the comprehension of parallel programs" *Proceedings of the Third Workshop on Program Comprehension*, 1994.

[8] Lencevicius, R.; Ran, A.; Yairi, R. "Third Eye - Specification-based Analysis of Software Execution Traces", *Proc. of the Int'l Conference on Software Engineering*, 2000.

[9] Luckham D.C. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison Wesley Professional, 2002.

[10] Luckham, D. C. and Frasca, B., "Complex Event Processing in Distributed Systems". *Stanford University Technical Report CSLTR --98—754*,March 1998.

[11] Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering 21(4)*, 1995.

[12] Moc, J.; Carr, D.A., "Understanding distributed systems via execution trace data", *Proc. of the 9th Int'l Workshop on Program Comprehension*, 2001.

[13] R.N. Taylor et. al. *A Component- and Message-Based Architectural Style for GUI Software*. IEEE Transactions on Software Engineering, June 1996.