An Approach for Tracing and Understanding Asynchronous 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 to humanly grasp the dynamic behavior of such applications, since many threads are active and events are asynchronously (and profusely) transmitted. We present a set of requirements for an aid to assist in the human understanding and exploration of the behavior of such applications through the incremental refinement of rules for determining causality relationships between messages sent among components. A prototype tool is presented, indicating one viable approach to meeting these requirements. Experience with the tool reinforces some of the requirements and indicates others.

1. Introduction

In event-based architectural styles, 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.

Understanding an event-based application without support tools and methods is difficult due to the large quantity of events flowing through an architecture, the complex, asynchronous interactions among components, and the lack of mechanisms within applications for understanding causal relationships between events. These interactions are so different from those of a tightly-coupled, synchronous system (e.g., most object-oriented systems) tools that work well on them, such as traditional program debuggers, usually work poorly on event-based applications. Debugging synchronous applications is done by examining program flow and variable values. However, at the architectural level it is the interaction that occurs *between* components in the form of events and messages that is of interest.

Black-box components, for which source code or specifications may be unavailable, present an additional difficulty. Thus, techniques are required that can determine causality relationships of messages and events that involve both black- and white-box components.

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 so events can be gathered for viewing and analysis?
- How can causal relationships between messages in an event-based architecture be determined?
- How can a user explore and refine the basis for causal relationships efficiently?
- How can messages be organized and visualized to cultivate an understanding of the system?

Experience with building and evolving event-based systems 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 complete formal behavioral specification.
- Event acquisition should minimally disturb application characteristics. Some effect on performance is expected, but semantic changes should be avoided.

Message Relationships and Causality:

- Causal relationships among messages must be determined without access to component source code to remain applicable to black-box components.

- Determination of causality relationships need not always be accurate, but any inaccuracies in identified 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.
- Tools should allow users to incrementally explore and refine the basis for determining causality relationships without the repeating previous steps in the process.
- Determination of causality should function even when component specifications are partial or in-progress.

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 present the data in a way that does not overwhelm a user.
- Tools that provide multiple views of data, and multiple methods of filtering those views, are preferable.
- Visualization tools should support a user's exploration and refinement of causality relationships by reflecting said changes immediately during exploration.

Approaches that work in environments with distributed or dynamic event-based systems are preferred. Additional goals we have identified but not fully explored 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.

2. Background

Architecture-based specification and analysis tools that deal with event-based systems such as Rapide [1] and CEP [2] are based on *complete* behavioral specifications of components. Furthermore, Rapide is designed for prototyping software architectures, and CEP does not specifically take into account architectural topology. Our approach applies to implemented, running systems, is architecture based, and does not require complete component specifications.

Many state-based analysis tools check whether a system meets its specification, usually expressed in statecharts or a similar formalism such as LTSA [3]. Many of these approaches focus on verification or analysis of early system artifacts rather than addressing a running system, e.g. [4]. In general, state-based analysis techniques suffer from state-explosion problems and are applicable only to small systems, even with significant effort to reduce the state space, as in Nitpick [5]. Our method of determining causality, in contrast to many state-based approaches, is far simpler than state machines, is not limited to simulations, and need not be complete.

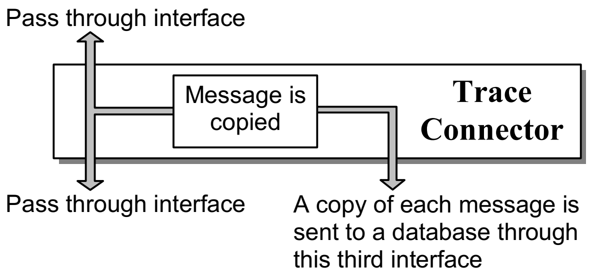


Figure 1. Structure and behavior of a trace connector.

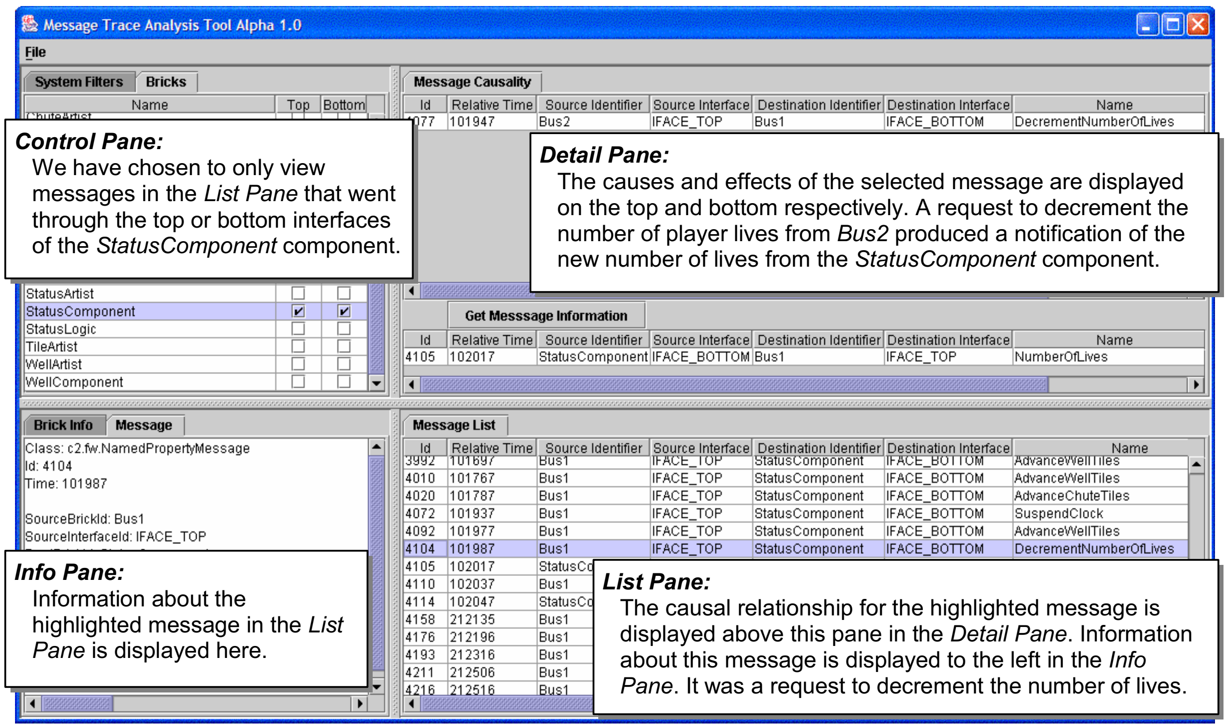
The distributed and parallel computing communities have addressed issues of event causality in concurrent, distributed programs [6]. In these communities, causality is primarily used to establish a global system snapshot, which is important for certain kinds of analysis. Our approach is focused on program understanding, so 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.

Monitoring and logging tools examine component interactions for large, commercially distributed applications facilitated by message-oriented middleware [7]. These tools typically diagnose and manage network problems and performance. 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 [8]. Furthermore, these tools do not address message causality, or allow the tracing of a sequence of messages through the distributed system.

Finally, tracing-based approaches have been used for debugging [9-11]. These approaches combine message tracing with traditional functions of a debugger. Most approaches augment traditional program debuggers, which require components' source code [10]. Our approach works with black-box components and allows a user to examine event traces that occurred at any time during a system, instead of interactively stepping through a system execution.

3. Approach

We developed an approach that meets the basic goals above and evaluated its feasibility through creation and application of prototype tools on representative event-based applications. Our approach uses a message trace to understand the communication among components in an architecture, rules to identify causality relationships be-



Control Pane:
We have chosen to only view messages in the *List Pane* that went through the top or bottom interfaces of the *StatusComponent* component.

Name	Top	Bottom
ChuteArtist		
StatusArtist	☐	☐
StatusComponent	✓	✓
StatusLogic	☐	☐
TileArtist	☐	☐
WellArtist	☐	☐
WellComponent	☐	☐

Id	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
4077	101947	Bus2	IFACE_TOP	Bus1	IFACE_BOTTOM	DecrementNumberOfLives

Detail Pane:
The causes and effects of the selected message are displayed on the top and bottom respectively. A request to decrement the number of player lives from *Bus2* produced a notification of the new number of lives from the *StatusComponent* component.

Id	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
4105	102017	StatusComponent	IFACE_BOTTOM	Bus1	IFACE_TOP	NumberOfLives

```
Class: c2.fw.NamedPropertyMessage
Id: 4104
Time: 101987

SourceBrickId: Bus1
SourceInterfaceId: IFACE_TOP
```

Info Pane:
Information about the highlighted message in the *List Pane* is displayed here.

Id	Relative Time	Source Identifier	Source Interface	Destination Identifier	Destination Interface	Name
3992	101697	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4010	101767	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4020	101787	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceChuteTiles
4072	101937	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	SuspendClock
4092	101977	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	AdvanceWellTiles
4104	101987	Bus1	IFACE_TOP	StatusComponent	IFACE_BOTTOM	DecrementNumberOfLives
4105	102017	StatusCo				
4110	102037	Bus1				
4114	102047	StatusCo				
4158	212135	Bus1				
4176	212196	Bus1				
4193	212316	Bus1				
4211	212506	Bus1				
4216	212516	Bus1				

List Pane:
The causal relationship for the highlighted message is displayed above this pane in the *Detail Pane*. Information about this message is displayed to the left in the *Info Pane*. It was a request to decrement the number of lives.

Figure 2. MTAT (Message Trace Analysis Tool) screenshot.

tween those messages, and a visualization tool to examine the chains of causality within a system.

Messages are captured by modifying the architectural description of a system rather than modifying components or using the underlying framework. This approach works with real, implemented systems that may contain black-box components.

Causal relationships between messages are determined using heuristic rules specified for each component. Heuristic rules may identify false causality relationships in addition to true ones, however, they are simpler to specify and can be applied to components where only partial behavioral specification is known.

A visualization tool allows the user to examine message causality chains of a running system using the specified rules as guides. During analysis, rules can be incrementally refined to better capture causality relationships and the changes will immediately be reflected in the tool.

3.1. Gathering architecture events

We collect message traces by automatically instrumenting an architecture with *trace connectors*. Trace connectors (see Figure 1) 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. Trace connectors are first-class entities, and are not part of any component in the architecture.

Our approach works in both single-process and distributed systems. In distributed systems, connectors bridging process boundaries are broken into two halves—one in each process. In terms of instrumentation, distributed connectors are treated the same way single-process connectors are; a trace connector is placed on every link in every process. When tracing message causality, distributed systems are viewed as unbroken architectures. Messages crossing multi-process connectors appear just as if they had crossed an in-process connector.

3.2. Determining causal relationships

In our approach, when a component or connector receives a message (or set of messages) and emits another message (or set of messages) in response, we say that the emitted messages are *caused* by the received messages. Without component source code, it is not possible to modify the component itself to tag each emitted message with a list of *caused-by* messages. An alternative would be to use complete formal models; though these can be impractical or impossible to create, especially for black-box components.

Instead, we have developed a simpler model of component behavior that indicates *probable* causes and effects of messages sent to/emitted by components with a high degree of certainty. We express causal message relationships using a simple language of *message classifiers* and *rules*.

3.2.1. Message Classifiers. Messages are classified by specifying message characteristics. A message is of-a particular type if it fits those characteristics. For example, a message classifier might be described informally as "any message with the name 'A'".

For our prototype tools, we assume that messages consist of a character string name and a set of name-value pair properties. Property names are character strings, and property values are arbitrary objects. We chose this format because it is general enough to be representative of many event-based applications, including those that were the target of our evaluation. However, other message formats and classifiers are feasible.

3.2.2. Rules. Rules describe, at varying levels of abstraction, how a component reacts to messages. They are typically specified by the software architect or component developer. Rules are *not* complete specifications of a component's behavior. Rules can be determined by viewing a component's source code (if available), by examining its documentation, or by knowledge of how the component responds to messages.

Rules are heuristic, meaning that they may occasionally indicate false causality relationships. In our experience, these inaccuracies had minimal practical impact, especially as rule accuracy is incrementally increased to exclude false relationships. Further analysis may indicate that this is true in the general case.

Each rule defines a set of *causes* and a set of *effects*. Causes and effects are specified as sets of message classifiers, the number of required occurrences for each message classifier, and the interface on which a message would be received or emitted.

3.2.3. Rule Types. There are different types of rules; the rule type indicates how the set of causes and effects are related. We have defined two rule types: *MatchingN* and *MostRecent*.

MatchingN rules indicate that a component will *always* send the complete set of effect messages when it receives a complete set of cause messages.

MostRecent, rules indicate that a component will respond to a set of events immediately or will not respond at all. This rule will associate the *most recent* complete set of cause messages with a complete set of effect messages.

While we believe that *MatchingN* and *MostRecent* rules capture many, if not most, of the types of causal relationships that occur in event-based systems, we intend to create additional rule types in the future. When determining causality, rules are translated into database queries that return possible causes and effects of a given message. Therefore, new rule types that can be translated to database queries are preferred.

3.3. Visualizing application events

MTAT (Message Trace Analysis Tool), a graphical tool that we have developed (see Figure 2), allows the user to interact with the message log and explore message causality chains, using the specified rules as guides. The database synchronizes the acquisition of messages from a running system with searches for causes among them, allowing the approach to be used interactively on a running system. MTAT also allows the rules to be changed while it runs, so rule changes are immediately reflected in its results.

MTAT allows a user to view the message log in its entirety, or to limit the display to messages of a particular component/interface combination. Users can also apply custom filters that 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 selected message's possible causes and its possible effects. Double-clicking a possible cause or effect changes the selected message, 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 causality chain.

Due to the heuristic nature of the approach, and possibly inaccurate rules, inaccurate results may be displayed. These will include false causes and effects, exclude true causes and effects, or both. It is always possible to determine whether the results are accurate by examining the unfiltered message log since it is complete. If an event did not occur and it should have, then this may indicate a bug in the component. If an event did occur, but the effects and causes are not listed, then there is likely a defect in the rule specification. 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.

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.

The first application, 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. We decided to trace KLAX because of its manageable size, and because we had no previous knowledge about the behavior of the game's components. We used our instrumentation tool to instrument the KLAX architecture and gathered approximately 40,000 messages from a single game of KLAX. Performance impact was not noticeable because the game speed is regulated by messages emitted by a clock component, thus the application has a significant amount of idle time when messages may be logged. We were able to trace causal chains of messages throughout the KLAX architecture using the MTAT tool, verifying our basic hypothesis and applicability of our rule language and tools.

The second application, the AWACS Simulator, is a distributed simulation of the software systems used on the US Air Force AWACS aircraft. It consists of 126 components and 206 connectors. We chose this system because its size is typical of large, "real-world" systems, it has an event-based architecture, and it is distributed across processes. When working with the AWACS message trace, we found that MTAT's performance when analyzing causal relationships was roughly equivalent to that found when tracing KLAX. 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). The AWACS experience indicates the scalability of our approach to large systems and verifies that our approach works when applied to distributed software systems.

5. Conclusion

The rule set used to specify causal relationships is simple, intuitive, and usable, but not fully formal, and applies to systems where component source may not be available. 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." In evaluation of we found that the actual number of false-positives was low and easy to determine when they did occur.

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. Acknowledgements

We would like to thank Adrita Bhor, Nghi Nguyen, and Santiago Li for their contributions to developing and evaluating this approach and its associated tools.

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 and the National Science Foundation under grant number 0205724. 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.

7. References

[1] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, pp. 336-355, 1995.

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

[3] S.-C. Cheung and J. Kramer, "Checking subsystem safety properties in compositional reachability analysis," presented at 18th International Conference on Software Engineering, Berlin Germany, 1996.

[4] J. M. Atlee and J. D. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *IEEE Transactions on Software Engineering*, vol. 19, pp. 24-40, 1993.

[5] D. Jackson and C. A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *IEEE Transactions on Software Engineering*, vol. 22, pp. 484-495, 1996.

[6] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, pp. 149-174, 1994.

[7] Software Engineering Institute, "Message-Oriented Middleware: Software Technology Review," vol. 2003: Software Engineering Institute, 2003.

[8] D. C. Luckham and B. Frasca, "Complex Event Processing in Distributed Systems," Stanford University, Stanford, Computer Systems Laboratory Technical CSL-TR-98-754, March 1998.

[9] A. P. Cláudio, M. B. Carmo, and J. D. Cunha, "Monitoring and Debugging Message Passing Applications with MPVisualizer," presented at 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodes, Greece, 2000.

[10] M. Frumkin, R. Hood, and L. Lopez, "Trace-driven debugging of message passing programs," presented at First Merged International Symposium on Parallel and Distributed Processing, Orlando, FL, 1998.

[11] R. Lencevicius, A. Ran, and R. Yairi, "Third eye — specification-based analysis of software execution traces," presented at 22nd international conference on Software engineering, Limerick, Ireland, 2000.