The Interaction Analyzer: A Tool for Debugging Ubiquitous Computing Applications

Nam Nguyen, Leonard Kleinrock, Peter Reiher
Computer Science Department, UCLA
Los Angeles, CA, USA
songuku@cs.ucla.edu, lk@cs.ucla.edu, reiher@cs.ucla.edu

Abstract—**Ubiquitous computing applications are frequently long-running and highly distributed, leading to bugs that only become apparent far from and long after their original point of appearance. Such bugs are hard to find. This paper describes the Interaction Analyzer, a debugging tool for ubiquitous computing applications that addresses this problem. The Interaction Analyzer uses protocol definitions and histories of executions that displayed bad behavior to assist developers in quickly finding the original root cause of the bug. We describe the architecture of the tool and the methods it uses to rapidly narrow in on bugs. We also report overheads associated with the tool, simulation studies of its ability to find bugs rapidly, and case studies of its use in finding bugs in a real ubiquitous computing application.**

Keywords-ubiquitous computing; debugging

I. INTRODUCTION

Ubiquitous and pervasive computing systems are often complex systems consisting of many different objects, components and agents, interacting in complicated and unpredictable ways. The real world frequently intrudes into pervasive systems, adding to their unpredictability. As a result, such systems can frequently display unexpected, and often erroneous, behaviors. The size and complexity of the systems and their interactions make it difficult for developers to determine why these unexpected behaviors occurred, which in turn makes it difficult to fix the problems [1], [2], [3].

We built a system called the Interaction Analyzer to help developers of complex ubiquitous computing systems understand their systems' behaviors and find and fix bugs. The Interaction Analyzer gathers data from test runs of an application. When unexpected behavior occurs, it uses the data from that run and information provided during system development to guide developers to the root cause of errors. The Interaction Analyzer carefully selects events in the execution and recommends that the human developers more carefully examine them. In real cases, the Interaction Analyzer has guided ubiquitous application developers to the root cause of system bugs while only requiring them to investigate a handful of events. In one case, the Interaction Analyzer helped developers find a race condition that they were previously unable to track down; the entire debugging process took less than five minutes, while previously developers had spent several days unsuccessfully tracking the bug using more traditional debugging techniques.

In this paper, we describe how the Interaction Analyzer works and give both simulation results of its efficiency in tracking bugs and cases where it found real bugs in a real ubiquitous application. Section II describes the Panoply system, for which the Interaction Analyzer was built, and introduces the example ubiquitous. Section III describes the Interaction Analyzer's basic design and architecture. Section IV provides simulation results and real case studies. This section also includes basic overhead costs for the Interaction Analyzer. Section V discusses related work. Section VI presents our conclusions.

II. PANOPLY AND THE SMART PARTY

The Interaction Analyzer was built as part of the Panoply project. Panoply is a middleware framework to support ubiquitous computing applications. This paper is not primarily about Panoply itself, so only issues relevant to the Interaction Analyzer will be discussed here. More details on Panoply can be found in [4].

Panoply enables the simple creation, configuration, and discovery of computational contexts that support communication-based groups, location-based groups, and interest- and task-based groups. These groups, called spheres of influence, organize related peers, and scope communication and configuration. Panoply provides primitives for setting up controlled communications among ubiquitous computing application elements. For the purpose of understanding the Interaction Analyzer, one can regard Panoply as a support system for applications made up of discrete, but interacting, components at various physical locations. These components communicate by message, and generally run code in response to the arrival of a message. Code can also be running continuously or periodically, or can be triggered by other events, such as a sensor observing a real-world event.

Several applications have been built for Panoply, and the Interaction Analyzer has been used to investigate many of them. Due to space restrictions, we will limit our discussion of the Interaction Analyzer's use to one Panoply application, the Smart Party [5].

In the Smart Party, a group of people attend a gathering hosted at someone's home. Each person carries a small mobile device that stores its owner's music preferences and song collection. The party environment consists of a series of rooms, each equipped with speakers. The home is covered by one or more wireless access points.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

As each guest arrives, his mobile device automatically associates with the correct network to connect it to the Smart Party infrastructure. As party attendees move within the party environment, each room programs an audio playlist based on the communal music preferences of the current room occupants and the content they have brought to the party. Guests automatically and dynamically collaborate with the host network, which manages their collective preferences and steers the music choices. As guests move from room to room, each room's playlist adjusts to the current occupants and their preferences.

The Smart Party can fail in many ways. It can overlook users, or it can localize them into the wrong rooms. It can fail to obtain preferences from some users. Its algorithms for song selection can be flawed, resulting in endless repetitions of the same song. It can unfairly disadvantage some users in the selection. These are just a few of the many possible causes of failures. Because it must take into account user mobility, and even the possibility of users leaving the Smart Party in the middle of any operation, flawed code to handle dynamics can lead to multiple problems. These characteristics, which caused a good deal of difficulty in getting the Smart Party to operate properly, are actually likely to be common to a wide range of ubiquitous computing applications. Therefore, the Smart Party is a good representative example of the complexities of debugging a ubiquitous computing application.

The problems we actually encountered during the development of the Smart Party application included music playing in rooms with no occupants, failure of some Smart Party components to join the application, and race conditions that sometimes caused no music to play when it should. These and other bugs in the Smart Party were attacked with the Interaction Analyzer. The results will be presented in Section IV.

III. THE INTERACTION ANALYZER

A. Basic Design Assumption

The Interaction Analyzer was designed to help developers debug their applications. Therefore, it was built with certain assumptions:

- The source code for the application is available and can be altered to provide useful information that the Interaction Analyzer requires.
- The system was not for use during actual application deployment. Thus, we could assume more capable devices than might be available in real use, and did not need to fix problems in working environments.
- Knowledgeable developers would be available to use the recommendations of the Interaction Analyzer to find bugs. The Interaction Analyzer does not to pinpoint the exact semantic cause of a bug, but guides developers in quickly finding the element of the system, hardware or software, that was the root cause of the observed problem.

The Interaction Analyzer works on applications that have been specially instrumented to gather information that will prove useful in the debugging process. This instrumented application is run in a testing environment, gathering data as the application runs. When developers observe a bug that they need to diagnose, they stop the application and invoke the Interaction Analyzer on the information that has been saved during the run.

The instrumented code is wrapped by a conditional statement that checks the value of a predefined boolean constant. By altering this value, the instrumented code can be easily removed in the final release of the binary.

B. Protocol Definitions and Execution Histories

The Interaction Analyzer is organized around a *protocol definition* (which specifies how the application is expected to work) and an *execution history* (which describes what actually happened in the run of the application). Each of these is a directed graph of *events*, where an event corresponds to some interesting activity in the execution of the system. Developers instrument their code to indicate when events occur and to store important information about those events. An event can be primitive or high-level. High-level events are typically composed of one or more primitive events, under the control of the developer.

The Interaction Analyzer uses both temporal order and causal order (such as sending a message necessarily preceding its receipt) of events to build the execution history of an application's run. Some of these relationships are found automatically by the Interaction Analyzer's examination of the source code, while some must be provided explicitly by the developers using instrumentation tools. By recording all events and their causal relationships that occur during the execution of a system, one can reconstruct the image and the detailed behavior of the running system at any time [6].

The protocol definition describes how the system should react and behave in different situations. We store the protocol definition in event causality graph format. The protocol definition is produced at design time, and the execution history is produced at run time.

C. Creating the Protocol Definition

The protocol definition is a model of the application's expected behavior. Such modeling is always an essential part of a large software project, and is helpful in smaller projects, as well. Models help software developers ensure that the program design supports many desirable characteristics, including scalability and robustness [7]. The Interaction Analyzer requires developers to perform such modeling using UML, a popular language for program modeling. We added some additional elements to the standard UML to support the Interaction Analyzer's needs, such as definitions of protocol events and relation definitions. We modified a popular graphical UML tool, ArgoUML [8], to create a tool called Argo-Analyzer that helps developers build their protocol definition.

The details of Argo-Analyzer are extensive. Briefly, developers use this tool to specify an application's objects, the relationships between them, the context, and the kinds of events that can occur in a run of the application.

The application is organized into objects. Object types are defined using Argo-Analyzer. For source code written in OOP languages (such as Java), the classes correspond to the object types. These object definitions are used to organize the protocol definition and describe interactions between different application elements.

Relationship definitions describe relationships between objects. Argo-Analyzer supports popular relationships such as parent-child, as well as other user-defined relationships.

Event templates define the properties of an instance of an important event in the application. There must be an event template for each type of event in the application. The Interaction Analyzer will use these templates to match an execution event with an event in the protocol definition.

The developer uses these and a few other UML-based elements to specify the protocol definition, which describes how he expects his application to work. This definition is, in essence, a directed graph describing causal chains of events that are expected to occur in the application.

Serious effort is required to create the protocol definition, but it is a part of the overall modeling effort that well-designed programs should go through. As with any modeling effort, the model might not match the actual instantiation of the application. In such cases, an execution history will not match the protocol definition, requiring the developer to correct one or the other. In practice, we found that it was not difficult to build protocol definitions for applications like the Smart Party, and did not run into serious problems with incorrect protocol definitions. Mismatches between definitions and executions were generally signs of implementation bugs.

D. Creating the Execution History

There is one protocol definition for any application, but each execution of that application creates its own execution history. The Interaction Analyzer helps direct users to bugs by comparing the execution history for the actual run to the expected execution.

The execution history is gathered by instrumenting the application. We provide a library to help with this process. This general-purpose Java library provides an interface to generate different kinds of events and their important attributes and parameters. An application generates an entry in its execution history by calling a method in this library. Doing so logs the entry into a trace file on the local machine. Applications can also define their own kinds of events, which the library can also log.

A typical analyzer record contains several fields, including a unique ID for the event being recorded, a developer-defined ID, information on the producer and consumer of the event (such as the sender and receiver of a message for a message-send event), timestamps, and various parameters specific to the particular kind of event being recorded. Most of the parameters are defined by the application developers, who can also add more parameters if the standard set does not meet their needs.

Adding the code required to record an analyzer event costs about the same amount of effort as adding a printf statement to a C program.

Panoply applications run on virtual machines, one or more on each participating physical machine. Each virtual machine can run multiple threads, and each thread can generate and log execution events to a local repository using the Event Analyzer's Execution History Generator component. When a run is halted, the Log Provider component on each participating physical machine gathers the portion of the execution history from its local virtual machines and sends it to a single Log Collector process running on a centralized machine. When all logs from all machines have been collected, the Log Collector collates them into a single execution history.

E. Using the Interaction Analyzer

After developers have created the protocol definition, instrumented their code to build the execution history, and run the instrumented application, they may observe bugs or unexpected behaviors during testing. This is when the Interaction Analyzer becomes useful. Upon observing behavior of this kind, the developer can halt the application, gather the execution logs (with the help of the Log Collector), and then feed them into the Interaction Analyzer. This graphical tool will then allow the developers to obtain answers to a number of useful debugging questions, including:

1. Why did an event E not occur?
2. Why did an incorrect event E occur?
3. What are the differences in behavior between objects of the same type?
4. Why did an interaction take a long time?

Each of these types of questions requires somewhat different support from the Interaction Analyzer. We will concentrate on how it addresses questions of Type 1 and 2.

1) Type 1 Questions

Type 1 questions are about why something did not happen when it should have. For example, in the Smart Party, if a user is standing in one of the rooms of the party and no music is playing there at all, developers want to know "why is no music playing in that room?" There are several possible reasons for this bug. Perhaps the user is not recognized as being in that room. Perhaps the user's device failed to receive a request to provide his music preferences. Perhaps the room was unable to download a copy of the chosen song from wherever it was stored.

The Interaction Analyzer handles Type 1 questions by comparing the protocol definition and the execution history to generate possible explanations for the missing event. The protocol definition describes event sequences that could cause an instance of that event. The execution history shows the set of events that actually happened, and usually contains partial sequences of events matching the sequences derived from the protocol history. The Interaction Analyzer determines which missing event or events could have led to the execution of the event that should have happened. These sequences are presented to the developer, ordered by a heuristic. The heuristic currently used for presenting possible descriptions of missing events is, following Occam's Razor, to suggest the shortest sequence of missing events first. The developer

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

examines the proposed sequence to determine if it explains the missing event. If not, the Interaction Analyzer suggests the next shortest sequence.

As a simplified example, say that music is not playing in a room in the Smart Party when guests are present there. The missing event is thus "play music in this room." The Interaction Analyzer will compare the sequence of events in the actual execution where music did not play to the protocol definition. It might come up with several hypotheses for why music did not play. For example, perhaps the guest who had selected a song failed to send it to the player. Or the module that gathers suggestions might have failed to ask any present guests for recommendations. Or the guests might not have been properly recognized as being in that room at all. The first of these three explanations requires the fewest "missing events" to serve as an explanation, so it would be investigated first.

The actual methods used by the Interaction Analyzer are more complex [9], since links in the protocol definition and execution history can have AND and OR relationships. Also, the Interaction Analyzer makes use of contextual information defined in the protocol definition and recorded in the execution history. For example, if a Smart Party supports music played in several different rooms, a question about why music did not play in the living room will not be matched by events that occurred in the kitchen.

2) Type 2 Questions

Type 2 questions are about why an incorrect event occurred. In the Smart Party context, such questions might be "why was Bill localized in the dining room instead of the family room" or "why did music play in the entry hall when no one was there?" Type 2 questions are thus about events that appear in the execution history, but are seen by the developer to either not belong in the history, or to have some incorrect elements about their execution.

The Interaction Analyzer works on the assumption that errors do not arise from nowhere. At some point, an event in the application went awry, due to hardware or software failures. The Interaction Analyzer further assumes that incorrectness spreads along causal chains, so the events caused by an incorrect event are likely to be incorrect themselves. If a developer determines some event to be incorrect, either that event itself created the error or one of the events causing it was also erroneous. Working back, a primal incorrect event caused a chain of incorrect events that ultimately caused the observed incorrect event. The developer must find that primal cause and fix the bug there.

Given these assumptions, the job of the Interaction Analyzer in assisting with Type 2 questions is to guide the developer to the primal source of error as quickly as possible. A standard way in which people debug problems in code is to work backwards from the place where the error is observed, line by line, routine by routine, event by event, until the primal error is found. However, this approach often requires the developer to check the correctness of many events. In situations where the execution of the program is distributed and complex (as it frequently is for ubiquitous applications), this technique may require the developer to analyze a very large number of events before he finds the actual cause of the error.

Is there a better alternative? If one has the resources that the Interaction Analyzer has, there is. The Interaction Analyzer has a complete trace of all events that occurred in the application, augmented by various parameter and contextual information. Thus, the Interaction Analyzer can quickly prune the execution history graph of all events that did not cause the observed erroneous event, directly or indirectly, leaving it with a graph of every event in the execution history that could possibly have contained the primal error. The question for the Interaction Analyzer is now: in what order should these events be analyzed so that the developer can most efficiently find that primal error?

Absent information about which events are more likely than others to have run erroneously, any event in this pruned graph is equally likely to be the source of the error. Assume this graph contains N events. The final event where the error was observed is not necessarily any more likely to be the true source of the error than any other, and, if the developer examines that event first and it was not the source of the error, only one of N possibly erroneous events has been eliminated from the graph. What if, instead, the Interaction Analyzer directs the developer to analyze some other event E chosen from the middle of the graph? If that event proves correct, then all events that caused it can be eliminated as the source of the primal error. Event E was correct, so the observed error could not have "flowed through" event E; thus the source of our error is not upstream of E. It must be either downstream or in some entirely different branch of the graph. If event E is erroneous, and E is one of the initial events of the application (one with no predecessor events in the graph), that E is identified as the root cause. If E is not one of the initial nodes, then it is on the path that led to the error, but is not necessarily the original cause of the error. We now repeat the algorithm, but with event E as the root of the graph, not the event that the developer originally observed, and we continue this process until we find the root cause.

With a little thought, one realizes that the ideal choice of the first event to suggest to the developer would be an event which, if it proves correct, eliminates half of the remaining graph from consideration. If no such event can be found, due to the shape of the graph, then choosing the event whose analysis will eliminate as close to half the graph as possible is the right choice.

This is the heuristic that the Interaction Analyzer uses. It prunes irrelevant events from the execution history graph and finds the event node in that graph whose elimination would most nearly divide the remaining graph in half. It then directs the developer to investigate that event. If the event proves correct, half the graph is eliminated, and the Interaction Analyzer then chooses another event using the same heuristic. If the event that the developer examines is erroneous, the Interaction Analyzer prunes the graph to use this erroneous event as the new root, and finds an event in the new graph to examine. Eventually, the highest erroneous event in the graph is identified as the root cause.

At each step, the developer manually investigates one event and tells the Interaction Analyzer whether that event is correct. But by using this technique, the developer need not work his way entirely up the whole execution history graph until he finds the problem. In general, the Interaction Analyzer allows the developer to perform the debugging with few human analysis steps. (In four real cases, using the Analyzer required 4-12 events to be examined, out of 200-21,000 total events, depending on the case.) As long as the Interaction Analyzer's automated activities (building the graph, analyzing it to find the next event to recommend, etc.) are significantly cheaper than a human analysis step, this process is much faster and cheaper than a more conventional debugging approach.

IV. USING THE INTERACTION ANALYZER

Here we present simulation results indicating how well the Interaction Analyzer would perform when working with execution graphs of different sizes. We also present case studies involving the actual use of the Interaction Analyzer in finding bugs in the Smart Party application, and data on the performance overheads of the system.

A. Simulation Results

To determine how the Interaction Analyzer would perform when handling large execution graphs, we generated artificial execution graphs of varying sizes and properties (such as the branching factors in the graph). Erroneous events and their root causes were generated randomly. The results are too extensive to report here (see [9] for full results), but one graph will give an enlightening picture of the actual benefits of using this tool, and the value of the algorithms it uses to find events for developers to examine.

When looking for a Type 2 error ("why did this incorrect event occur?"), one could examine the graph of all events that directly or indirectly caused the erroneous event and randomly choose one to examine. Unless some nodes are more likely to be erroneous than others, randomly selecting one of the nodes to examine is no more or no less likely to pinpoint the root case than walking back step-by-step from the observed error, which is a traditional debugging approach. For reasons not important to this discussion, we have termed the algorithm that randomly selects a node from the graph "Terminal-Walk."

The algorithm that the Interaction Analyzer actually uses (see Section III.E.2) analyzes the portion of the execution graph associated with the erroneous event and directs the developer to an event whose correctness status will essentially eliminate half the nodes in this graph. We term this approach the "Half-Walk" algorithm.

Figure 1 shows the performance of these algorithms for event graphs of different sizes. The x-axis parameter refers to the number of nodes in the causal graph rooted at the observed erroneous event, any one of which could be the root cause of the observed error. The x-axis is a log scale. The "Validation Cost" is in number of events, on average, that the developer will need to examine by hand to find the error. The graphs analyzed here have uniform branching factors and a uniform probability that any event in the graph rooted at the observed erroneous event (including that event itself) is the root cause of the error.

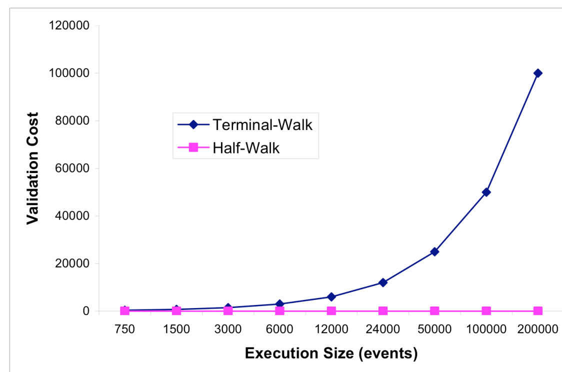

Figure 1. Terminal-Walk vs. Half-Walk Algorithm

The Terminal-Walk algorithm becomes expensive as the number of potential causes of the observed error grows. Each validation represents a human developer examining code and state information for an event in the system, which is likely to take at least a few minutes. The Half-Walk algorithm, on the other hand, is well behaved, displaying $\log_2$ behavior.

In some situations, the probability of failure in each event is known. For example, the system may consist of sensors with a known rate of reporting false information. Even if event failure probabilities are not perfectly known, an experienced developer's may have a sense of which events are likeliest to be the root cause of errors. If the developer has perfect knowledge of this kind, he will be able to instantly assign a probability of being erroneous to all events in the system. He might use an algorithm that first examines the event with the highest probability of being correct. If that event is indeed correct, he could eliminate from further consideration all events that caused that event. He could then move down the list of probabilities as candidates are eliminated. We term this algorithm the Highest-Walk algorithm.

Figure 2. Highest-Walk Algorithm vs. Half-Walk Algorithm

Figure 2 shows the relative performance for the Highest-Walk algorithm vs. the Half-Walk algorithm

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

(which the Interaction Analyzer actually uses) for graphs and root causes of the same kind shown in Figure 1. Highest-Walk is, unlike Terminal-Walk, competitive with Half-Walk, but Half-Walk is clearly better. For 200,000 events in an execution graph, Half-Walk will require the developer to examine less than half as many events as Highest-Walk would. The probability of being incorrect is propagated down the event path, and thus the event with highest probability of being incorrect is normally very far from the root cause. Thus, the Highest-Walk does not perform as well as Half-Walk.

B. *Case Studies Using the Interaction Analyzer*

Simulation studies are helpful in understanding the Interaction Analyzer's behavior in many different circumstances, but ultimately the point of a debugging tool is to prove helpful in debugging real problems. In this section we describe how the Interaction Analyzer helped us find real bugs in a real application, the Smart Party application we introduced in Section II. This application was not written to help us investigate the behavior of the Interaction Analyzer. On the contrary, the Interaction Analyzer was built to help us debug problems with the Smart Party and other Panoply applications.

1) *Music Playing in the Wrong Room*

This bug occurred in the Smart Party when a party was run with three rooms and one user. Music played in a room where no user was present. Before availability of the Interaction Analyzer, the developers of the Smart Party had used traditional methods to find the root cause of this problem, which proved to be that the module that determined a user's location had put him in the wrong room. We did not keep records of how long the debugging process took before the Interaction Analyzer was available, but it was far from instant.

This was a Type 2 error, an event occurring incorrectly. As mentioned in Section III, the Interaction Analyzer uses contextual information when available to guide the process of finding root causes. We investigated this bug both with and without contextual information. Without contextual information, the Analyzer had to suggest six events (out of a possible 8000 in the execution history) to pinpoint the problem. With contextual information (the developer indicating which room he was concerned about, which was not difficult to obtain), the Interaction Analyzer found the problem in one step.

2) *No Music Playing*

This bug occurred in some, but not all, runs of the Smart Party. A user would join the Smart Party, but no music would play anywhere. Since this bug was non-deterministic, it was extremely hard to find using standard methods. In fact, the Smart Party developers were unable to find the bug that way.

Once the Interaction Analyzer was available, it found the bug the first time it occurred. This was a Type 1 error, an event that did not occur when it should have. The Interaction Analyzer found the root cause by comparing the protocol definition to the execution history and noting a discrepancy. The Interaction Analyzer made use here of its ability to deal with events at multiple hierarchical levels. At the high level, it noted that music did not play and that the high-level protocol definition said it should. The Analyzer determined that the failure was due to not responding to a request by the user for a localization map. To further determine why that request wasn't honored, the Analyzer suggested to the developer that he dive down to a lower protocol level, and, eventually, to an even lower level. The bug ultimately proved to be in the code related to how Panoply routed messages.

The Interaction Analyzer found this bug in three queries, a process that took less than five minutes, including the time required by the developers to examine the code the Analyzer recommended they look at. The developers had been unable to find this bug without the Analyzer over the course of several weeks.

TABLE 1. INTERACTION ANALYZER COSTS

Operation	Example Cost	Average Cost
Import Exec Hist.	3.5 seconds	.35 msec/event
Preprocessing	.3 seconds	.03 msec/event
Load Prot. Def.	7 seconds	.82 msec/element
Matching	12.2 seconds	1.18 msec/event
Total Time	23.0 seconds	2.48 msec/event

3) *Interaction Analyzer Overheads*

Table 1 shows some of the overheads associated with using the Interaction Analyzer. The Example Cost column shows the actual total elapsed times for handling all events in a sample 11,000 event execution history. The Average Cost column shows the normalized costs averaged over 20 real execution histories. These costs are paid every time a developer runs the Interaction Analyzer, and essentially represent a startup cost. For an 11,000 event run, then, the developer needs to wait a bit less than half a minute before his investigations can start.

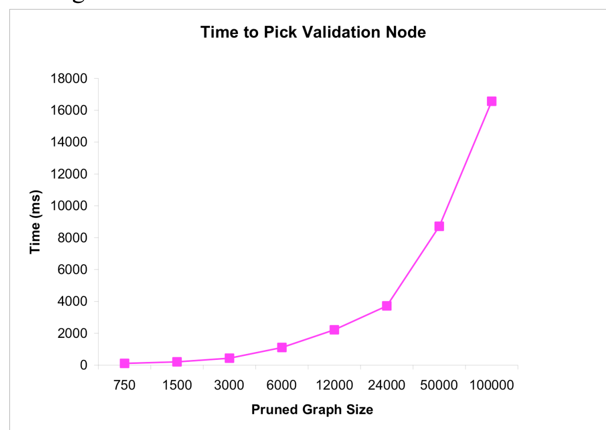

Figure 3. Time to Pick Validation Node

The other major overhead is the cost for the Interaction Analyzer to respond to a user query. For queries of Types 1, 3, and 4, this cost is less than a second. For queries of

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-171-7

Type 2, it depends on the size of the portion of the execution history that is rooted at the event the developer needs to investigate, not the size of the entire history. Any event that exerted a causal influence on the event under investigation must be considered. Figure 3 shows the time required to choose an event for the developer to evaluate for causal graphs of different sizes. If there are 100,000 events in the causal graph of the investigated event, it takes around 17 seconds to recommend one to the developer. This graph is log scale on the x-axis, so the time is roughly linear as the number of events grows. The Interaction Analyzer chooses an event for validation such that its examination will eliminate around half of the graph, so if the event in question is not the root cause, the second recommendation will be made on a graph of half the size of the original, and thus half the cost.

V. RELATED WORK

Several systems have supported debugging problems in complex distributed systems. The most closely related are those that build execution graphs based on data gathered during a run. RAPIDE [10] was an early system that used this approach, which was extended to build an execution architecture that captured causal relationships between runtime components [11]. The developers must manually examine the graph to identify the causes.

The Event Recognizer [12] matches actual system behavior from event stream instances to user-defined behavior models to assist in debugging. The goal is to find the mismatch and present it to the developers. Poutakidis et al. [13] uses interaction protocol specifications and Petri nets to detect interactions that do not follow the protocol.

Other approaches use non-graph-based methods to find root causes. Yemini and Kliger [14] treat a set of bad events as a code defining the problem, and uses decoding methods to match it to known problems. Piao [15] uses Bayesian network techniques to determine root causes of errors in ubiquitous systems. Ramanathan [16] and Urteaga [17] proposed systems for finding root causes of errors in sensor networks based on examining various metrics in those networks.

VI. CONCLUSIONS

Ubiquitous systems are complex, consisting of many different components. Their dynamic nature makes it hard to develop and debug them. Bugs often become evident long after and far away from their actual cause. The Interaction Analyzer provides quick, precise determination of root causes of bugs in such systems. While developed for Panoply, it can be adapted for many ubiquitous computing environments. The Interaction Analyzer has been demonstrated to have good performance by simulation, and has been used to find actual bugs in real ubiquitous computing environments, including cases where more traditional debugging methods failed.

ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation under Grant CNS 0427748.

REFERENCES

[1] W. Edwards and R. Grinter, "At Home With Ubiquitous Computing: Seven Challenges," LNCS, Vol. 2201/2001, 2001, pp. 256-272.

[2] J. Bruneau, W. Jouve, and C. Counsel,"DiaSim: A Parameterized Simulator for Pervasive Computing Applications," Mobiquitous 2009, pp. 1-3.

[3] T. Hansen, J. Bardram, and M. Soegaard, "Moving Out of the Lab: Deploying Pervasive Technologies in a Hospital," Pervasive Computing, Vol. 45, Issue 3, July-Sept. 2006, pp. 24-31.

[4] K. Eustice, Panoply: Active Middleware for Managing Ubiquitous Computing Interactions, Ph.D. dissertation, UCLA Computer Science Department, 2008.

[5] K. Eustice, V. Ramakrishna, N. Nguyen, and P. Reiher, The Smart Party: A Personalized Location-Aware Multimedia Experience, Consumer Communications and Networking Conference, January 2008, pp. 873-877.

[6] P. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," ACM TOCS, Vol. 13, No. 1, February 1995, pp. 1-31.

[7] The Object Management Group, http://www.omg.org, Sept. 2011.

[8] ArgoUML, the UML Modeling Tool. http://argouml.tigris.org, Sept. 2011.

[9] N. Nguyen, Interaction Analyzer: A Framework to Analyze Ubiquitous Systems, Ph.D. dissertation, UCLA Computer Science Department, 2009.

[10] D. Luckman and J. Vera, "An Event-Based Architecture Definition Language," IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 2005, pp. 717-734.

[11] J. Vera, L. Perrochon, and D. Luckham, "Event Based Execution Architectures for Dynamic Software Systems," IFIP Conference on Software Architecture, 1999, pp. 303-308

[12] P. Bates, "Debugging Heterogeneous Distibuted Systems Using Event-Based Models of Behaviors, ACM Transactions on Computer Systems, Vol. 13, No. 1, February 1995, pp. 1-31.

[13] D. Poutakidis, L. Padgham, and M. Winikoff, "Debugging Multi-Agent Systems Using Design Artifacts: The Case of Interaction Protocols, 1st International Joint Conference on Autonomous Agents and Multiagent Systems, 2002, pp. 960-967.

[14] A. Yemini and S. Kliger, "High Speed and Robust Event Correlation," IEEE Communications Magazine, Vol. 34, No. 5, May 1996, pp. 82-90.

[15] S. Piao, J. Park, and E. Lee, "Root Cause Analysis and Proactive Problem Prediction for Self-Healing," Int'l Conference on Convergence Information Technology, 2007, pp. 2085-2090.

[16] N. Ramanathan, et al., "Sympathy for the Sensor Network Debugger," Int'l Conference on Embedded Networked Sensor Systems, 2005, pp. 255-267.

[17] I. Urteaga, K. Barnhart, and Q. Han, "REDFLAG: A Runtime, Distributed, Flexible, Lightweight, and Generic Fault Detection Service for Data Driven Wireless Sensor Applications, Percom 2009, pp. 432-446.