Debugging Ubiquitous Computing Applications With the Interaction Analyzer

Nam Nguyen, Leonard Kleinrock, and 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 points of origin. Such bugs are difficult 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 a bug. We discuss characteristics of ubiquitous computing applications that can complicate debugging. We describe the architecture of the Interaction Analyzer 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 real ubiquitous computing applications.**

Keywords-ubiquitous computing; distributed debugging; ubiquitous applications

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][4].

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 [1]. 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 of an application 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 application. 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 and Section VI presents our conclusions.

II. PANOPLY AND THE SMART PARTY

The Interaction Analyzer was designed to be a general-purpose system usable in many ubiquitous computing contexts. However, since we wished to demonstrate its use in a real environment, we needed to connect it to some particular system. We chose to integrate the Interaction Analyzer with Panoply. Panoply is a middleware framework to support ubiquitous computing applications. While the work described here treats the Interaction Analyzer in the Panoply context, we should emphasize that, with relatively little effort, the Analyzer could be integrated with other types of ubiquitous computing system. The suitability and ease of the port will depend on the degree to which the target system relies on a message-based paradigm for interactions, since that is what the Analyzer itself expects.

While the key elements of the Interaction Analyzer do not depend on Panoply constructs, understanding how we used it in the Panoply context requires a little knowledge of how Panoply works. The core representational unit of Panoply is the *Sphere of Influence*, which can represent an individual device or a group of devices united by a common interest or attribute such as physical location, application, or social relationship. Spheres unify disparate notions of "groups," such as device clusters and social networks, by providing a common interface and a standard set of discovery and management primitives.

Panoply provides group management primitives that allow the creation and maintenance of spheres of influence, including discovery, joining, and cluster management. A publish/subscribe event model is used for intra- and inter-sphere communication. Events are propagated between devices and applications, subject to scoping constraints embedded in events of interest. Every sphere scopes policy

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and contains a policy manager [5] that monitors the environment, mediates interactions and negotiates agreements.

Panoply supports the design of applications that express their needs and communicate through events. Panoply applications (e.g., the Smart Party) can create custom events, and designate the scope and destination of such events. More details on Panoply can be found in [6].

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 [5], [6], [7], and the Interaction Analyzer has been used to investigate many of them. We will concentrate our discussion of the Interaction Analyzer's use on one Panoply application, the Smart Party [7], touching more lightly on its use for other applications.

In the Smart Party, a group of people attends 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. **Figure 1** shows the configured version of a Panoply Smart Party, in which three rooms in a house are capable of playing music and party attendees with various different musical preferences are located in each room.

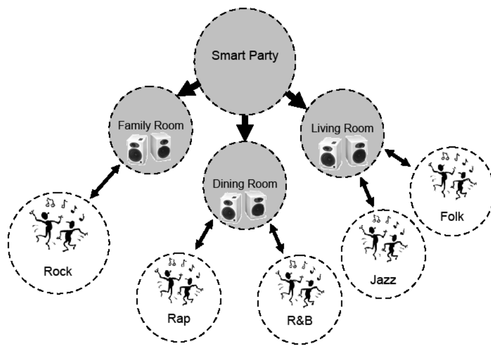

Figure 1. A Panoply Smart Party

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. For example, for the party in **Figure 1**, rock music would play in the family room, since the guests there all have that preference, while folk or jazz would play in the living room.

A Smart Party room determines which guests are present because they have enrolled automatically in a Panoply sphere belonging to that room, triggered by wireless network enrollment. The Panoply sphere controlling the Smart Party in that room periodically queries the devices of the users in that room for their music preferences. These preferences are currently expressed as particular songs the user would like to hear played. The Panoply sphere then uses the combined responses and a voting procedure [8] to select a song from among those suggested by the users' devices. That song is downloaded to the room (from the user's device or somewhere he specifies) and played, after which the process repeats.

As guests move from room to room, the underlying Panoply framework notices their movements and removes them from their old room, adding them to the new one. Thus, each room's playlist adjusts to the current occupants and their preferences.

The Smart Party is a real, working application, extensively tested in our labs.

The Smart Party application could fail in many ways. It could overlook users, or it could localize them into the wrong rooms. It could fail to obtain preferences from some users. Its algorithms for song selection could be flawed, resulting in endless repetitions of the same song. It could 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 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 such applications.

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 intended for use during application development, not ordinary application use. This assumption allowed us to rely on the presence of more capable devices (with greater

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

storage capacity and processing power, for example) than might be available in real deployment.
- Knowledgeable developers would be available to use the recommendations of the Interaction Analyzer to find bugs. The Interaction Analyzer does not pinpoint the exact semantic cause of a bug, but guides developers in quickly finding the element of the system, hardware or software, that is the root cause of the observed problem. Also, this assumption meant that we did not need to provide descriptions of problems that would be meaningful to naïve users unfamiliar with Panoply or the design of the application.

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. The data is stored and organized automatically for use during debugging, if necessary. 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 Interaction Analyzer is not intended to find bugs on its own. Rather, it assists developers in finding and understanding the causes of observable bad or unexpected application behaviors. The Interaction Analyzer is not intended as a replacement for tools that perform automated analysis of source code, but as a tool for diagnosing problems with application behavior.

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.

The Interaction Analyzer was designed for use in a Linux environment, and is implemented in C. It was designed for debugging programs written in C or C++. It could be ported to other environments with reasonable ease.

B. Protocol Definitions and Execution Histories

The Interaction Analyzer uses 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) to debug applications. 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, as specified by the developer.

The Interaction Analyzer uses both temporal order (one event occurring before another) and causal order (such as the event that sends a message must precede the event that receives the message) 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 others must be provided explicitly by the developers using instrumentation tools. By recording all events that occur during the execution of a system and their causal relationships, one can reconstruct the image and the detailed behavior of the running system at any time [9].

The protocol definition describes how the system should react and behave in different situations. We store the protocol definition in an 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 [10]. 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 [11], to create a tool called Argo-Analyzer that helps developers build their protocol definition.

The Argo-Analyzer is itself a complex system. 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 the 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. The Argo-Analyzer supports commonly used 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. For an event to match a template, not only must their event type and parameter fields match, but their causality requirements must also match. If the execution event does not have the same kinds of preceding events as the template, it will not match.

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

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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, which generally should be fixed even if they do not instantly cause incorrect behavior.

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 in particular runs of the application by comparing the execution history for that run to the expected execution described in the protocol definition.

Each event in the application should generate a record in the execution history. There are three ways to collect the system information required to create such records that describe an execution history of a program: software, hardware and hybrid. For the Interaction Analyzer, software monitoring was used since it provides more flexibility and does not need extra hardware support.

The monitoring could have been based on external observation of the application's behavior, which would have had the advantage of not requiring any instrumentation of the application. We would have needed some way to observe the scheduling of events, which would have involved observing messages being sent between objects in the system. Inter-machine messages could have been sniffed off the wire (though use of cryptography would have complicated this approach). Messages that did not cross real network boundaries would have been more challenging to capture. In either case, obtaining information about the state of the sending and receiving objects would have been difficult.

We chose instead to gather information about the execution history by instrumenting the application. This approach provides greater detail and produces more powerful execution traces than external monitoring could provide. It does so at the cost of changing the application source code. However, since the target use of the Interaction Analyzer is by application developers during their development and debugging process, the costs were of less concern, and the benefits more compelling.

We provide a library to help with this instrumentation process. This general-purpose Java library provides an interface to generate different kinds of event records 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. Panoply itself logs its own special kind of events, such as "sphere joins," using this mechanism.

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, pointers to all events that directly caused this record's event, 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. For example, the command to generate an event in the Smart Party application under Panoply looks similar to this:

PanoplyLogger.logPanoplyObjectCreated(codeID ,panoply-specificEvent,creator,createdObject, directCauses,additionalParameters);

Compare this to an actual print statement that the Smart Party developers used before the Interaction Analyzer was available:

Debug.println(ModuleName,Debug.DETAIL, panoply-specificEvent.EventType + " "+panoply-specificEvent.EventSubType+ " " +additionalMessage);

The two statements are of similar length and complexity, and require that the developers provide roughly the same information. However, the old version merely allowed a message to be printed, while the Interaction Analyzer version allowed much more, as will be discussed later.

Typically, all statements that record information on the execution history for the Interaction Analyzer are bracketed by compiler commands to include or exclude them, depending on a compile-time option. Thus, a final recompilation when debugging is finished produces a version of the code without any overheads related to the recording of event history.

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 its portion of the execution history from its local virtual machines and sends this history 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 merged 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 are likely to 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

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

application, gather the execution logs (with the help of the Log Collector), and feed them into the Interaction Analyzer. The Interaction Analyzer makes use of both the protocol definition and the execution history.

The Interaction Analyzer is a graphical tool that was built using an internal frame model where the main window contains multiple sub-frame-like windows of two types:

1. System-type windows: These windows are created by default and support the major functions of the Interaction Analyzer.

2. User-type windows: These windows are created (and destroyed) by the developer who is using the Interaction Analyzer for debugging. Typically, each user-type window contains information about particular events or objects in the protocol definition or the execution history.

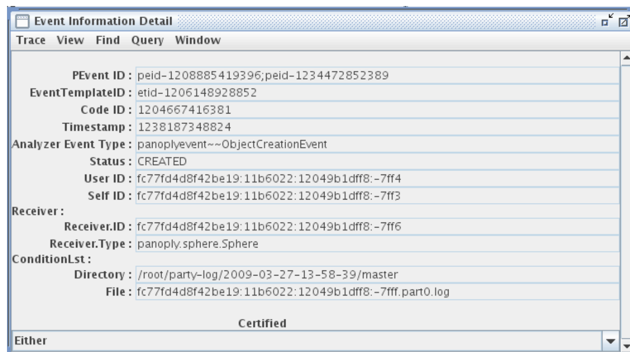

Figure 2. Screenshot of the Interaction Analyzer

Figure 2 is a snapshot of what use of the Interaction Analyzer looks like when the developer starts it. At this point, no execution history has yet been loaded, so all the windows are generic to the application in the abstract, rather than being specific to the erroneous run being debugged. Using a menu option, the user would choose the execution history describing the buggy run he wishes to analyze, at which point the windows would become populated with information specific to that run, and the developer could start to work.

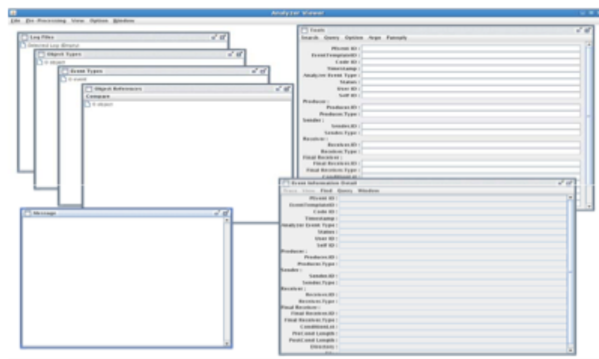

Figure 3. A Sample User-Type Window for an Event

As debugging proceeds, the developer opens and closes windows and navigates between them to assist in tracking down the problem he has observed. **Figure 3** is an example of one user-type window that describes an event from the execution history. In this case, it is a Panoply event that has created a sphere. The window shows various event parameter values, such as when the event occurred and what type it was. The Interaction Analyzer will suggest events that are particularly likely to be helpful in debugging various problems, and the user performing the debugging might open this window to help him determine whether there was an obvious error in this particular event.

When a developer opens such an event window, he can take various actions. For example, if the Interaction Analyzer has suggested that this event might be the cause of the error, the developer can investigate the event and, if he determines it is correct, he can validate the event. That action tells the Interaction Analyzer that it should offer a different event as the possible cause. Alternately, the user can ask to see upstream events, perhaps because he suspects that the error that was observed here originated further back in the execution trace, or because he needs more context to understand what should be going on in this event. He can view events at different hierarchical levels, diving down for more detail or popping up to see a higher-level picture of the sequence of events. Similarly, he can ask for downstream events to see what this event led to.

Another option is to find the matching event description in the protocol definition. This option would allow the developer to compare what the protocol said should happen to what actually occurred for this event. Protocol events are described by a similar window, and allow similar kinds of actions: navigation forward and backward, changing of hierarchical levels, obtaining more detail, and so on.

The Interaction Analyzer allows 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?

The developer asks these questions from one of the system-type windows created when the Interaction Analyzer starts execution. For example, to ask a question of Type 1, the developer would specify the event ID of the protocol event he expected to see, but did not, in a field in the Tools system-type window, which is the window in the upper right of **Figure 2**.

Each of the types of questions that a developer can ask requires somewhat different support from the Interaction Analyzer. We will concentrate on how it addresses questions of Type 1 and 2. The Interaction Analyzer also supports searching for particular execution events and protocol events.

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

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 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 developer performing the debugging will ask a Type 1 question focused on why the "play music" event did not occur in this instance. Complicating factors include the fact that, in the same run, music might have been properly played in other rooms, or even previously or subsequently in the room in question. Thus, the Interaction Analyzer offers methods of specifying the particular context in which debugging should proceed. In this case, the context is the room where the music didn't play, at the moment when silence was noticed.

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 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.

There are many possible approaches to determining the relevance of different possible explanations, which then guides where the Interaction Analyzer directs the developer. As mentioned, the Interaction Analyzer currently chooses the explanation with the shortest path, where the path length is defined as the number of events to be added or removed to resolve the problem. In this example, the first of these three explanations (that the guest failed to send the song to the player) 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 [12], since links in the protocol definition and execution history can have AND and OR relationships. Further, real protocols tend to be multilayered and complex. In the case under discussion, for example, sub-protocols are used for user localization, voting, and file transfer. The error could have arisen in any one of these lower-level protocols, in which case eventually the developer would need to move down from the high-level protocol that deals with Smart Party concepts, like asking users for music preferences, to the low-level protocol that might control the transfer of a large file from one or several places to a destination. The Interaction Analyzer understands the concept of multi-layer protocols and offers tools for navigating up and down through these layers.

Further, 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. The developer performing the debugging will need to specify the context he cares about, since the Interaction Analyzer itself does not know that an event that should have occurred in the living room did not, and thus cannot specify that the location context is the living room. As the developer navigates through the execution graph using the Interaction Analyzer's advice, he is able to refine his search with further contextual information.

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 that the developer feels do not belong in the history, or 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 that some event is 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, event by event, routine by routine, eventually line by line, until the primal error is found. However, this approach often requires the developer to

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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? Eventually, the developer will need to perform some amount of detailed analysis of code and execution data, but can the Interaction Analyzer help to minimize how much of that analysis is required?

In the absence of information about which events are more likely than others to have run erroneously (which is generally the case), 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. After all, one of its predecessor events could easily have run erroneously, with the error propagating and only being discovered at this point. If the developer examines the observed incorrect 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), then 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 can then 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 such an event proves incorrect, it eliminates the other half of the graph, since either this event or something upstream must be the root cause, not anything downstream. (There is an assumption here that errors do not simply "go away." Thus, if we are examining event E because an erroneous event downstream of E was observed, and the event E is also erroneous, the Interaction Analyzer assumes that the path of error flowed through event E.) If no event whose examination can eliminate half the graph can be found, due to the shape of the graph, then selecting the event whose analysis will eliminate as close to half of the graph as possible is the right choice.

The Interaction Analyzer uses this heuristic to select events for developer investigation. After pruning irrelevant events from the execution history graph, it directs the developer to investigate the event node in that graph whose elimination would most nearly divide the remaining graph in half. The algorithm proceeds as suggested above, eliminating roughly half of the remaining nodes at each step, and eventually the highest erroneous event in the graph is identified as the root cause.

The algorithm stops when it finds this event. It assumes that the incorrectness of the event causes the observed problem that led to debugging, and thus it is the root cause. That is not necessarily true. Consider event X caused by events Y and Z. We observe some incorrect behavior in X, examine Y, and determine it is incorrect. The algorithm would report Y as the root cause, if nothing upstream of Y is incorrect. However, it's possible that the real problem is in event Z, which the algorithm never suggested examining.

However, if the incorrectness of the event reported did not actually cause the observed problem, then it is the root cause of a different problem. In other words, this algorithm will find the root cause of either the original problem detected by the developer or another problem that he does not know of a priori. If the developer can determine that this event, despite being incorrect, could not have caused the observed behavior, he can run the Analyzer again, this time indicating that this incorrect event actually is correct. This will cause the Analyzer to look elsewhere, and thus to find a different root cause of the original problem. In the example above, having determined that, despite its incorrectness, event Y could not have caused the observed misbehavior in event X, a second run of the Interaction Analyzer with Y marked as correct would lead us to event Z, the true cause of the problem.

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 examination of 4-12 events, out of 200-21,000 total events, depending on the case. Some detailed examples will be discussed in Section IV B.) 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 less expensive than a more conventional debugging approach.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

IV. EVALUATING THE INTERACTION ANALYZER

There are several aspects to the performance of the Interaction Analyzer that should be addressed in determining its value. We outline them here.

The Interaction Analyzer will be of most value in diagnosing problems in large complex ubiquitous environments where many events and possible causes of problems muddy the waters. In such situations, though, there are obvious questions about whether the Interaction Analyzer can perform sufficiently well at that scale. It will need to gather and analyze a great deal of information, and perhaps the costs of doing so will be too high for practical use. Therefore, we performed various simulation experiments to investigate the Interaction Analyzer's performance on large execution graphs.

Ultimately, the Interaction Analyzer is valuable if it can actually help ubiquitous computing application developers find tricky problems in their systems. To demonstrate the Interaction Analyzer's promise in meeting this fundamental goal, we present case studies involving the actual use of the Interaction Analyzer in finding bugs in the Smart Party application and a second smaller application.

A secondary, but important, practicality aspect of the Interaction Analyzer is its basic performance overheads, so we present data on those overheads, as well.

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 completely here (see [12] for full results), but some example graphs will show the actual benefits of using this tool and the value of the algorithms it uses to find events for developers to examine.

For each point in the simulation figures shown here, 20 different rooted execution graphs were generated randomly. For each generated graph, 10 different scenarios were generated randomly for different root cause nodes. For a given execution graph and root cause, most of the tested algorithms are deterministic, except for the Terminal Walk algorithm (described below). Even for that algorithm, only one simulation was performed for a given graph and root cause. Thus, each point in the figures represents 200 different runs. The value at each point in the figures is the mean value of the number of validation requests for all scenarios. The size of the graph in all simulations represents the rooted graph size. In other words, we assume all irrelevant events that could not directly or indirectly cause the detected incorrect event are already pruned from the graph. The confidence level is 95%. In some cases, the variation is small enough that the error bars are essentially invisible.

A major question for the Interaction Analyzer is how it chooses which event to suggest for further investigation. 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 recommend for examination. Unless some nodes are more likely to be erroneous than others, randomly selecting one of the nodes to examine is just as likely to pinpoint the root case as 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." Results for the Terminal-Walk algorithm thus also describe the traditional approach, under the assumption that any event in the pruned graph is equally likely to be the original source of error.

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, as described earlier. We term this approach the "Half-Walk" algorithm.

Figure 4 shows the performance of these algorithms for event graphs of different sizes. The x-axis parameter refers to the number of nodes in the pruned 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 first graphs we discuss have uniform branching factors (i.e., the branching factor of each node in the graph is randomly chosen from a uniform distribution, with each branch indicating an event caused by the event described by this node). The probability that each node in the graph (including the node where the error was observed) is the root cause of the error is generated from a uniform distribution. The actual root cause is then randomly selected following that probability distribution. The Terminal-Walk algorithm is unaware of this probability distribution, and merely randomly selects one of the possible events from the graph for examination.

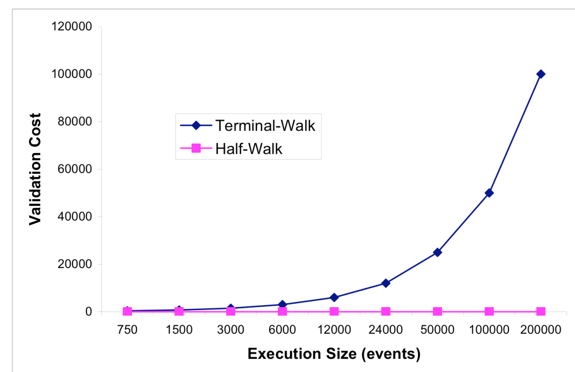

Figure 4. Terminal-Walk vs. Half-Walk Algorithm

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 may have a sense of which events are likeliest to be the root cause of errors.

If the developer has perfect knowledge of the probability that each event was performed correctly, 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 5 shows the relative performance for the Highest-Walk algorithm vs. the Half-Walk algorithm (which the Interaction Analyzer actually uses) for graphs and root causes of the same kind shown in **Figure 4**. If we have this knowledge of probability of event failure, the Half-Walk algorithm is altered so that, instead of selecting an event that eliminates half the nodes in the graph, it selects an event that eliminates half the probability of error in the graph. 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 algorithm does not perform as well as Half-Walk.

Figure 5. Highest-Walk Algorithm vs. Half-Walk Algorithm

Many factors can cause variation in the performances of these algorithms. For example, the distribution of branching factors can be varied, where branching factor describes how many events are caused by a given event. The figures already discussed assumed a branching factor for each node chosen from a random distribution. Perhaps, instead, there is a distribution of branching factors randomly generated in a linear scale, where nodes with a lower branching degree appear more often in the event graph. Thus, branching factors of 1 are more probable than branching factors of 2, which are more probable than branching factors of 3, and so on.

Figure 6 illustrates the relative performance of the Highest-Walk and Half-Walk algorithms in this case. Since the distribution used here favors small branching factors, generally a node will have a lower branching factor in these graphs than in those discussed earlier. Here, the Highest-Walk performs much worse than the Half-Walk because the Highest-Walk chooses the node with highest probability of being correct for validation. Consequently, after each validation, the new size of the event graph (in the Highest-Walk) tends to be large. In other words, it prunes the tree less effectively. With uniform branching (as in **Figure 5**), the graph is more balanced; whereas in linear branching, the graph is less balanced. With uniform branching, we are less likely to find nodes with a really high probability of being incorrect. Thus, in uniform branching, the Highest-Walk has better performance, though it is still inferior to the Half-Walk algorithm.

Figure 6. Constant Root Cause Factor – Linear Branch

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 that it prove helpful in solving real problems. In this section we describe how the Interaction Analyzer helped us find real bugs in real applications. The primary application discussed is the Smart Party application introduced in Section II. This application was not written to help us investigate the behavior of the Interaction Analyzer. On

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the contrary, the Interaction Analyzer was built to help us debug problems with the Smart Party and other Panoply applications.

We also present a case study for a second real application, one that is much smaller. Our graduate students grew tired of having to get up to open the locked door to our lab when someone without a key knocked on it, so they designed a bit of hardware and a small Panoply application that would automatically unlock the door under certain conditions. For example, if a knock was heard during regular working hours and someone was actually in the lab, the door should open. Also, users in the room with sufficient privilege could simply order the door to open. This application was vastly simpler than the Smart Party, but it was also real working Panoply software, and exhibited real bugs.

1) Smart Party Bug 1: Music Playing in the Wrong Room

This bug occurred in the Smart Party when the application 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 user location determination module 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), the Interaction Analyzer found the problem in one step.

2) Smart Party Bug 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 difficult 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. Without the Analyzer, the developers had been unable to find this bug over the course of several weeks.

3) Door Opener Bug

The door opener application described previously had a bug that caused the door not to open when it was ordered to do so. Since the bug appeared to be failure of an expected event, a query of Type 1 was used, and the Analyzer was able to find two different root causes. The first cause was a serial port configuration problem, and thus the application failed to open the port and could not send commands to the hardware controlling the door lock. The second cause was a mistake in the policy describing which spheres should talk to each other, and thus the command could not reach the controller. The first cause was found immediately since it happened in the highest protocol layer. The second root cause was found after going down five protocol layers. There were about 340 execution events in the log files. The Interaction Analyzer recommended that a total of 6 events be examined to find both problems. This case demonstrates both the value of the Interaction Analyzer even for relatively simple applications, and how the Interaction Analyzer can find multiple bugs that cause a single observed problem.

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

C. 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.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 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 7** 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.

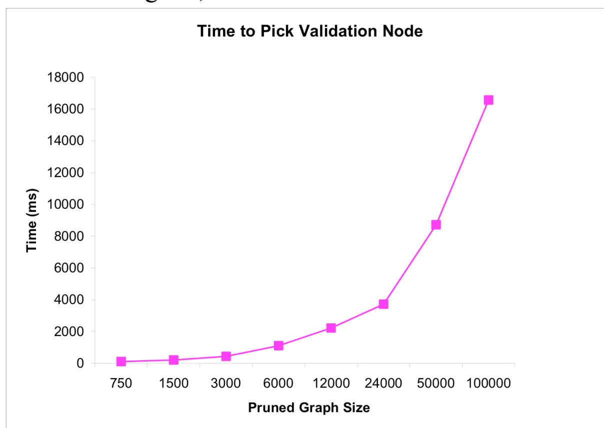

Figure 7. Time to Pick Validation Node

D. *Usability and Utility Issues*

The goal of the Interaction Analyzer is to ease the burden for developers, so whether it does so is a major concern. We have anecdotal evidence to support its value. In terms of cost, there is a learning curve associated with first using the system and with instrumenting applications. This curve is not steep, in our experience. The actual cost of instrumenting applications is relatively low. The statements that must be inserted into the code are no more complex than more traditional print debugging statements that most programmers already use (see Section III.E).

In terms of run time costs, the Interaction Analyzer is not intended to be run in production mode, so the performance costs are paid only when performing debugging. They are not sufficiently high to slow debugging, since they are no greater than printing log outputs. There is a cost to gather and analyze the data from a run, but this is a matter of a few seconds to a few minutes for a typical run (see Section IV.C).

While these costs are low, they are still not worthwhile unless there is commensurate benefit. Our experiences show that there is, as outlined by the case studies above. The Interaction Analyzer succeeded in finding real bugs that standard debugging techniques could not, even in situations where analysts had spent days or weeks tracking the bugs down. The Interaction Analyzer did so in a matter of a few minutes. It was helpful in finding complex bugs that were non-deterministic and depended on race conditions, situations that are notoriously hard to deal with using standard debugging techniques.

A fuller statement of utility and usability would require detailed human studies and experiments that have not been performed. However, the initial results are promising and suggest that the system is worth far more than its costs.

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 [13] was an early system that used this approach. In RAPIDE, an event can be used to denote any action or behavior of a system. By capturing enough events, the image of the application runtime can be reproduced later. The author also proposed hierarchical viewing for event management. RAPIDE aggregates sets of low-level system events into a higher-level event to give information about the component objects at the application level. Different abstraction hierarchies can be used to display the system in different views. RAPIDE also supports event filtering based on a predefined pattern.

RAPIDE was extended to build an execution architecture that captured causal relationships between runtime components [14]. This system creates an image of the running system that helps the developer visualize all interactions and relationships between components during execution. Based on the visual graph, developers can understand the execution architectures of dynamically changing software systems. If the execution architecture is different from the specification, an exception is raised to report the abnormality. This work supports building models of bad behavior to detect known problems.

The Event Recognizer [15] treats debugging as a process of creating models of expected program behaviors and comparing these to the actual behaviors exhibited by the program. The Event Recognizer matches actual system behavior from event stream instances to user-defined behavior models. Incompletely recognized behaviors indicate that the modeler should more closely examine the class of behaviors that are missing, or explain what is wrong with a particular program execution. The tool helps to detect abnormal behavior (that is not defined in the behavior models) and shows how well the actual behavior fits the user-defined pattern. If a bad behavior happens to fit one of the defined behavior models, the system will not be able to detect the problem.

Poutakidis et al. [16] uses interaction protocol specifications to detect interactions that do not follow the

protocol. Interaction protocols are specified using AUML and translated to Petri nets. The debugger uses Petri nets to monitor conversations and detect any unexpected or missing message when interaction does not follow the protocols. More specifically, it can detect failures such as un-initialized agents, sending messages to the wrong recipient, sending the wrong message and sending the same message multiple times. However, it does not explain why the problem occurs and the root cause.

Other approaches use non-graph-based methods to find root causes. Yemini and Kliger [17] treat a set of bad events as a code defining the problem, and use decoding methods to match it to known problems. This approach assumes that the developers know which sets of bad events occur when a problem happens.

Piao [18] uses Bayesian network techniques to determine root causes of errors in ubiquitous systems. The Bayesian network describes a complex system as a compact model that presents probabilistic dependency relationships between various factors in a domain. System real-time performance data is collected, including the system health states. Certain parameters are selected and ranked in a node list that will be applied in structure learning. Bayesian machine learning is applied for topology structure learning to find a network structure that is the most probable match to the training data. This network structure is used to infer the root cause for real-time data from an erroneous run. This approach requires a large training set to build a complete dependency graph and does not work well for a system with a large number of parameters due to the over-fitting phenomenon in machine learning. Therefore, it is more applicable for systems with small sets of parameters, such as a network system described solely by CPU, throughput, RAM use, and bandwidth.

Ramanathan [19] designed a system to find the root causes of errors in sensor networks. This system collects network-related data such as routing table, neighbor list, up time, bad packets received, etc. Based on the specific relationship between these collected data, the system detects failures and triggers localization. For example, the neighbor list and the up time can be used to detect a failed sensor node. Since the collected metrics are sensor-network specific, this approach can only be directly applied for sensor network environments.

Urteaga's REDFLAG system [20] is a fault detection service for data-driven wireless sensor applications. It consists of a Sensor Reading Validity (SRV) sub-service, which detects erroneous sensor readings, and a Network Status Report (NSR) subservice, whose task is to abate data loss by identifying unresponsive nodes. These two subservices help to identify failed sensors in the network.

Gardner [21] proposed a framework to monitor efficiently all system events and information in an operating system, with the goal of providing a detailed look at operating system kernel events with very low overhead. This collected information can be used to analyze problems in the underlying system and provide necessary information so that adaptive applications can adjust. This framework instruments the kernel and network library code to generate events; and the developers are expected to examine these recorded events by themselves to identify the problem. Also, this system is not distributed, and thus not easily adaptable for a ubiquitous environment.

Hseush's debugging approach [22] concentrates on data, rather than events, arguing that data is a more meaningful way for program users to approach debugging than control flow. In this approach, the user must be aware of data flow as well as control flow and/or message flow. A debugging language is provided to express breakpoints, single stepping and traces in terms of the data as well as the control. For example, concurrent accesses on a shared memory location can be described using the language; and when the user detects a matched behavior to the described concurrent accesses, the application can be stopped or suspended accordingly.

Other researchers have approached debugging of ubiquitous environments and other distributed systems from entirely different angles, potentially offering complementary ways to help diagnose problems in such environments. [23] describes a suite of tools that help visualize multi-agent applications. Each tool provides a different perspective of the application being visualized. For example, a society tool shows the structural organizational relationships and message interchanges between agents in a society, while a report tool graphs the society-wide decomposition of tasks and the execution states of the various sub-tasks. The complete set of tools provides various perspectives on the condition of the distributed application.

[24] and [25] propose different system views that allow a graphical representation of the selected aspects of the system state and its dynamic behavior. An agent view shows the structural or behavioral agent model. An interaction view shows patterns of interactions such as message-passing activity. A cooperation view shows the potential or current task requests between agents. These views represent the developer's conceptual models, such as the agent, distribution and interaction models proposed by the authors.

Such alternate approaches can conceptually be combined with the debugging services offered by the Interaction Analyzer, giving different perspectives from which to view any particular debugging problem. Which set of views and tools is most helpful for actual debugging of complex ubiquitous computing problems is a question for further research.

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

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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.

The Interaction Analyzer is fundamentally a tool to help developers perfect their ubiquitous computing applications. With significantly more work, it could perhaps be adapted to work in deployment scenarios, helping average users fix the problems they observe. Generally, however, the advice the Interaction Analyzer can provide would not be very helpful to a typical user. Much more effort would be required to assist in mapping from low-level problems to solutions that make sense to a typical user. Further, if the problem is rooted in flawed code, rather than a failed hardware component or a mistake in configuration, it is not likely that the user will be able to solve the problem, even if he can pinpoint it. Nonetheless, debugging for ubiquitous computing users is likely to be a problem of increasing importance for the future, and deserves more study.

ACKNOWLEDGMENT

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

REFERENCES

[1] N. Nguyen, L. Kleinrock, and P. Reiher, “The Interaction Analyzer: A Tool for Debugging Ubiquitous Computing Applications,” Ubicomm 2011, November 2011.

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

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

[4] 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.

[5] V. Ramakrishna, K. Eustice, and P. Reiher, "Negotiating Agreements Using Policies in Ubiquitous Computing Scenarios," Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA'07).

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

[7] 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.

[8] K. Eustice, A. Jourabchi, J. Stoops, and P. Reiher, “Improving User Satisfaction in a Ubiquitous Computing Application,” SAUCE 2008, October 2008.

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

[10] The Object Management Group, http://www.omg.org, July 12, 2012.

[11] ArgoUML, the UML Modeling Tool. http://argouml.tigris.org, July 12, 2012.

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

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the Sensor Network Debugger,” Int’l Conference on Embedded Networked Sensor Systems, 2005, pp. 255-267.

[20] 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.

[21] M. Gardner, W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz. “MAGNET: a Tool for Debugging, Analyzing and Adapting Computing Systems,” Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, 2003.

[22] W. Hseush and G. Kaiser, “Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language,” Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, Wisconsin, United States, May 1988.

[23] D. Ndumu, H. Nwana, L. Lee, and J. Collis. “Visualizing and Debugging Distributed Multi-agent Systems,” Autonomous Agents, Seattle WA, USA, 1999.

[24] M. Liedekerke and N. Avouris, “Debugging Multi-agent Systems,” Information and Software Technology, 1995.

[25] M. Morris, “Visualization for Causal Debugging and System Awareness in a Ubiquitous Computing Environment,” Adjunct Proceedings of UbiComp, 2004.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org