Ensuring Privacy in Presence Awareness Systems: An Automated Verification Approach

Patrice Godefroid†, James D. Herbsleb†, Lalita Jategaonkar Jagadeesan†, Du Li‡

†Bell Labs, Lucent Technologies
263 Shuman Blvd.
Naperville, IL 60566
{god, herbsleb, lalita}@research.bell-labs.com

‡Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024
lidu@cs.ucla.edu

ABSTRACT

Providing information about other users and their activities is a central function of many collaborative applications. The data that provide this "presence awareness" are usually automatically generated and highly dynamic. For example, services such as AOL Instant Messenger allow users to observe the status of one another and to initiate and participate in chat sessions. As such services become more powerful, privacy and security issues regarding access to sensitive user data become critical. Two key software engineering challenges arise in this context:

- Policies regarding access to data in collaborative applications have subtle complexities, and must be easily modifiable during a collaboration.
- Users must be able to have a high degree of confidence that the implementations of these policies are correct.

In this paper, we propose a framework that uses an automated verification approach to ensure that such systems conform to complex policies. Our approach takes advantage of VeriSoft, a recent tool for systematically testing implementations of concurrent systems, and is applicable to a wide variety of specification and development platforms for collaborative applications. We illustrate the key features of our framework by applying it to the development of a presence awareness system.

Keywords

Computer-supported Cooperative Work, Presence Awareness, Privacy, Security, Verification, Coordination

INTRODUCTION

Many CSCW application domains create specific and difficult software engineering problems. Ubiquitous computing, for example, presents a number of specific issues for the development of toolkits, rapid prototyping, and integration [1]. New software engineering challenges are presented by technologies designed to support distance collaboration, which are becoming increasingly common and important. AOL Instant Messenger, for example, now claims 80 million users who send 750 million messages a day [30]. It allows users to create "buddy lists" of people with whom they frequently communicate. Each time a user logs on, he or she is informed of everyone on their buddy list who is also logged on. The user can then initiate a chat with any of these "buddies." The application depends critically on users receiving information about the presence of others on the system.

Presence Information

This ability to convey "presence awareness" is rapidly becoming an important component of many collaborative applications (e.g., [23, 25]). For example, many calendar programs support sharing, so that coworkers can see when others are busy, or even see the contents of others' calendar entries. This detailed information can be extremely valuable for quickly locating colleagues, or for reviewing agendas of ongoing meetings to stay abreast of current issues and decisions [24].

Awareness information is particularly important for large, global organizations. In geographically distributed software development, for example, one of the most serious problems is the time it takes to resolve issues involving people at more than one site [14]. Presence information could go a long way toward alleviating problems like phone tag by informing distant colleagues about who is actually available, and when.

Many kinds of data can be used by current applications for presence awareness purposes, including whether a person is logged on (as in AOL Instant Messenger), audio and video of varying resolution (e.g. [31]), location information [29], and information about current environment (e.g., has the screen saver engaged, what web site is the user currently browsing, what file is currently being edited). Many kinds of "presence" data are generated automatically by the user's activities. Applications will be able to take advantage of additional sources of information as networks move toward convergence, i.e., carrying telephony and video, as well as data from networked applications. Nearly any detectable event could conceivably find a legitimate use in some "presence aware" application.

Privacy Concerns

From the users' point of view, there is a fundamental tradeoff between access to presence data for legitimate uses, and concerns about privacy (e.g., [17]). Precisely to the extent that I am able to identify what you are doing, I can communicate with you when the need arises, make my communications more timely and convenient for you, and generally be a more effective colleague. This is the sort of information, however, that users would generally not like to provide to strangers, nor perhaps to managers, or to competitors. Moreover, the data are largely generated automatically and potentially quite frequently, so users cannot be expected to monitor all presence events in order to ensure appropriate levels of privacy.

There are similar issues, of course, in managing access privileges more generally, for example in a file system. While implementations vary considerably, a good solution is conceptually quite simple. Each user has control over their own data, and the ability to determine what, if anything, is available to other individuals, groups, or even roles [26]. In a similar way, one could specify the availability of the various types of "presence" data. Presence awareness data in collaborative applications, however, presents several software engineering challenges that evade such solutions.

Software Engineering Challenges

Complexity and Modifiability of Policies Policies regarding access to data in collaborative applications have subtle complexities, and the data to which they are applied are highly dynamic. For example, applications such as NetMeeting allow everyone in an application-sharing collaborative session to see many windows open on one user's (user 1) desktop. It may be the case that this user has permission to view certain "presence" data about another user (user 2), for example, current phone activity, current location, calendar entries, or the contents of a chat window. If this is displayed on user 1's desktop, it may inadvertently be made available to everyone else in the application sharing session, even if the other participants do not have the correct access privileges. One might call this a violation of a "non-transitivity" policy.

Another example of the subtleties of these policies comes from the practice of "lurking," i.e., listening and/or viewing without otherwise participating, and perhaps without others' knowledge. This may often be desirable, for example, allowing students to observe ongoing scientific data collection and analysis activities and the accompanying conversation first hand (e.g., [22]). In other situations, it may be considered rude [15] or even threatening to allow a user to acquire data about other users that he or she does not permit them to acquire about him or her. One would like to be able to define a policy, for example, of "mandatory reciprocity," which would allow user 1 to access information concerning user 2 only if user 1 allows user 2 to access the same data about user 1. Other types of reciprocity may also be desirable (e.g., [25]).

Policy specifications must be easily modifiable during a collaboration. In general, it is impossible to specify the "correct" policy in advance of actual use [13] of a given collaborative application. Builders of collaborative tools must generally be able to try out various policies and quickly adjust them to suit the sensitivities of their user communities. Further, the desired policies may change rapidly as users gain more experience with the system, as, for example, trials with location-tracking badges [29] have shown. There may also be considerable variation in the policies desired by different groups of users within a single company, as research on shared calendars has demonstrated [24]. Moreover, one would like to be able to add people to a session, change the level of permissions, and so on, without concerns that one's privacy will be violated.

Correctness of Policy Implementations Users must be able to have a high degree of confidence that the implementations of these policies are correct. Otherwise, users will abandon an application very quickly.

Presence awareness systems, by their very nature, are highly concurrent, in that users are typically represented as concurrent elements. Such systems are extremely difficult to design and test because their components may interact in many unexpected ways. To ensure privacy in such systems, traditional testing techniques are of limited help, since test coverage is bound to be only a minute fraction of the possible behaviors of the system. Hence, these techniques do not provide sufficient confidence of the enforcement of complex user policies. As [Bullock and Benford, pp. 148] noted, it is difficult to demonstrate security, since it "involve[s] the long, drawn-out task of manually or semi-automatically inspecting a possibly complex environment to ensure there is confidence [in the system] . . . "

On the other hand, since the presence awareness data in these systems is very sensitive, users will not tolerate any security violations. Moreover, there will be great reluctance to use such systems if guarantees about the privacy of presence awareness information cannot be provided.

Our Approach

We propose a framework that uses an automated verification approach to ensure that such systems conform to complex policies that may be dynamically modified during a collaboration. Our verification framework can detect violations of complex policies using VeriSoft, a tool for systematically testing concurrent systems [11], and through run-time monitoring. We demonstrate the feasibility of our approach by analyzing a prototype collaboration tool, which is essentially an extended version of popular instant messaging systems incorporating additional types of presence data. For our feasibility evaluation, we built an implementation of this collaboration tool in COCA (Collaborative Objects Coordination Architecture)[18], a framework that supports the rapid, modular specification and modification of policies. Our automated analysis of this implementation using VeriSoft revealed some important and subtle errors in the system, which would have been virtually impossible to detect through traditional testing methods.

The remainder of the paper is as follows. We first describe the capabilities of a general presence awareness system that illustrates the kinds of actions that users may perform and the kinds of policies that they may specify. We then describe a possible architecture for such a system, together with some alternatives. We then present the properties we wish to verify of such a system. We next describe VeriSoft and the approach toward systematic testing of these kinds of properties. Our feasibility evaluation, including an example of error found by our analysis, is then discussed. We finish with our conclusions and future work.

CAPABILITIES OF A GENERAL PRESENCE AWARENESS SYSTEM

The presence awareness system we consider is based on existing systems such as AOL Instant Messenger, but extends them in two important ways:

1. It allows users to enquire about more sophisticated (and sensitive) kinds of presence information about others.
2. It allows users to dynamically specify their presence awareness policies, in order to control others' access to their own presence information.

Our presence awareness system supports the following types of user activities.

Users may update their own presence information

Presence information about a user may be updated explicitly through user actions, or implicitly through sensors. A user may explicitly update his presence information by logging in or logging out. The willingness to interact is iconified, for example, by the state of a door on the user's screen. An implicit presence awareness change may occur through sensors which detect and report the time-varying activities of a user, e.g. GPS (Global Positioning System) for user location tracking. In our prototype system, we model implicit presence awareness changes through a screen saver which reports user screen activities. For example, when a user has not been actively using input devices (e.g. mouse and keyboard) for a given period of time, a screensaver comes on. When the user touches some input device after a period of inactivity, the screensaver goes off. The screen saver off and on events are automatically generated by the user interface.

Users may enquire about the presence of other users

For example, users may be interested in:

- Login status of a user X, e.g. is X currently logged on and since when?
- Screen saver status of X, e.g. is X's screen saver on or off, and since when?
- Is X currently in a chat? Who are the other participants? How long has he or she been chatting?
- What is the door status of X? What are the access rules and settings of X?
- What is X's calendar, location, phone number, email address, etc.?

In some situations, e.g. a group discussion, anonymous participation often encourages contributions. All participants are anonymous by default, i.e., after a user X logs in, other participants only see a pseudo name unless the disclosure of X's real name is allowed explicitly by user X. Thus, users may be interested in finding out the real names of others.

Users may communicate with other users

We currently model user communication through multi-party text chat, since it is representative of many other forms of collaborative communication. Users may initiate a chat session, invite others to join an existing session, request to participate in an existing session, accept or decline others' requests to join a session, or leave a session. Once a user becomes a participant in a chat session, he or she can send messages which will appear on other participants' screen. Customized admission control policies dictate the rules for joining sessions; for example, it may require the session initiator's consent, or a vote of all participants which shows the consent of the majority. Multiple chat sessions can be active simultaneously across the Presence Awareness system users.

Users may set/modify their presence awareness policies

Inspired by [3, 10, 21], our system gives a user the capability to express his willingness to engage in casual interactions and to control which part of his private data can be accessed by whom. If the door is set open, then the user is ready for chat invitations from any other users. If otherwise the door is closed, then in principle this user should not be interrupted by any chat invitations.

Exceptions exist however, and are critical for collaborations. For example, in a closely-coupled work team, a user may give (some of) his colleagues the privilege to interrupt him even if his door is closed. Queries regarding the private data of a user – for example whether the user is available, what is the real name of the user, the recent chat activities of the user – can also be explicitly allowed or disallowed. All these motivate the specification of exception rules which are of the following form

$$set(Condition \rightarrow Action)$$

where $Condition$ is a boolean expression and $Action$ is an action expression. Intuitively, if $Condition$ evaluates to true then $Action$ is enabled. In what follows, $p^{<i>}$ denotes that user i can take action p, while $\neg\, p^{<i>}$ denotes that user i cannot take action p. For example, the rule "door(closed) $\rightarrow$ invite$^{<j>}$" means that, even if the door of the user in question, say i, is closed, user j can send a chat invitation to i. Rule "true $\rightarrow\ \neg$ check(name, pseudo(i))$^{<j>}$" means that under no circumstances should user j check the real name of user i by i's pseudonym. The right of access can be granted to individuals, set of designated users, user groups [27], even sets of users dynamically decided by a predicate[7]. In many frameworks, users can define exceptions to general policies. In AREA [9], for example, awareness of actions on artifacts is generally restricted to a specific set of awareness "situations." This policy can be overridden, however, to provide access to related materials, by defining an exception. To do so, a user must write a specification that includes an event description, a scope, a situation, and an admission list of actors. A collection of such specifications could become quite complex.

To economize, one often defines implicit rules and explicit rules instead. For example, the implicit rule is, when the door of a user say X is open, *in general*, any other users can send X a chat invitation; and when X's door is closed, *in general*, nobody can send X the invitation. To explicitly exclude a user say j from sending an invitation to user i even if i's door is open, we can set the following exception rule

$$\text{door}^{<i>}(\text{open}) \rightarrow\ \neg\ \text{invite}^{<j>}$$

And to explicitly grant j the permission to invite i even if i's door is closed, we set exception rule

$$\text{door}^{<i>}(\text{closed}) \rightarrow \text{invite}^{<j>}$$

A more sophisticated policy, reciprocal permission, may say that j can invite i when i's door is closed given that i can invite j; a similar policy can be specified for finding out real names instead of pseudonyms.

ARCHITECTURE OF THE PRESENCE AWARENESS SYSTEM

We now describe a possible overall architecture of this system, depicted in Figure 1.

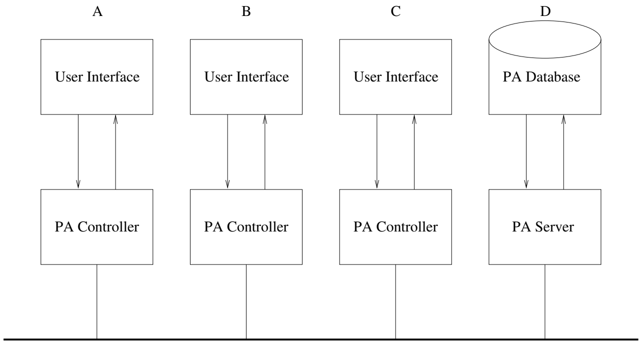

Figure 1: Four participants in a Collaboration.

Each participant in the presence awareness system has a user interface and Presence Awareness Controller (PA Controller) running at his site. Each participant interacts with the system through his user interface. In addition, there is a centralized Presence Awareness Database (PA Database) that stores awareness information such as users' private data, user activities, and awareness preferences settings. The Presence Awareness Server (PA Server) controls access to the Presence Awareness Database. In Figure 1, three participants, A, B, and C are in the "PA Controller" role. Participant D is in the "PA Server" role which controls the access to the database. All the entities in the same collaboration are connected by a collaboration bus. Those buses contain channels which communicate messages between connected entities. The term *gate* is used to denote access points to channels.

Specifying coordination policies is equivalent to defining the behavior of these two roles (PA Controller and PA Server) and how they coordinate with each other. Each such controller (or server) has three gates: g_{in} is used to receive messages from its user interface (or database), g_{out} to send messages to its user interface (or database), and g_{remote} to communicate messages with other controllers/server. Since channel g_{remote} is defined on the collaboration bus, it can be used to both send and receive.

User Interface

The user interface has four functions, corresponding to the types of user activities described in detail earlier. Each participant's user interface exchanges messages with its Presence Awareness Controller. In particular, the user interface may send the following messages to the rest of the Presence Awareness System:

login, logout, screensaver(on), screensaver(off),
check-availability(X), check-name(X), check-chatters(X),
set(door(open)), set(door(closed)), set(ExceptionRule)

The set(ExceptionRule) message corresponds to a user's request to set or modify his presence awareness policies. The other messages correspond to the activities described earlier.

The messages received by the user interface from the rest of the Presence Awareness System are:

available(X), unavailable(X), name(real(X),pseudo(Y)),
chatters(SID, SetOfChatters)

where each chat session is identified by a globally-unique id "SID" (see below).

In addition, the user interface may send or receive messages from the rest of the Presence Awareness System:

invite(SID, Inviter, Invitees), apply(SID, X),
accept(X, SID), reject(X, SID), message(SID, X,Text),
leave(SID, X)

Presence Awareness Controller

The Presence Awareness Controller specifies how a user interacts with other users through the Presence Awareness System. A PA Controller communicates and coordinates with other PA Controllers and the PA Server for the user it represents.

Presence Awareness Database and Server

In our prototype system, the Presence Awareness Database and Server form a centralized database which stores all awareness information, such as all users' private data, users' activities, and awareness preferences settings. More specifically, the following data are included for all users: (1) The time a user is logged on and logged out. (2) The time a user's screen saver is activated and deactivated. (3) The start time and end time of a chat session, who initiates chat, who are involved in a chat, who received the chat invitation but did not accept. (4) A user's accessibility settings, for example, the user's willingness to engage in interactions (door status), who is allowed to see which part of the user private data, who can check the user's availability, who can only see the availability conclusion and who can see how the conclusion is reached as well, exception rules, and so forth.

When a user logs on to the Presence Awareness system, his previous anonymity and access settings are retrieved from the database. All the availability checking and notifications are performed by the database. For example, the four messages login, logout, screensaver(on), and screensaver(off) modify the availability of a participant. These messages are sent from a user interface to its PA Controller, which passes them along to the the PA Server.

Upon receipt of any of these four messages, the PA Server queries the PA Database to determine the availability of the participant in question. In particular, a user is said to be available if and only if the user has not logged out since the last login, and either the screen saver has not gone on since the last time it went off or it has always been off.

The PA Server stores the latest status of a user, and then notifies all participants of any update to anyone's availability status in accordance with the users' policies. When a user X wants to check the availability of a user Y, the PA Controller for user X passes the request message to the PA Server. After querying the database, the PA Server sends an available(Y) or unavailable(Y) message to X, depending on the actual availability of Y and in accordance with the policies of X and Y.

Alternative Architectures

Alternative architectures may differ in two significant ways:

- Communication among entities may not involve a collaboration bus; for example, communication may be point-to-point.
- Presence awareness data may be distributed among the PA Controllers, rather than being stored in centralized PA Database.

EXAMPLES OF PROPERTIES

We now discuss properties that such a system must satisfy in order to be "correct". The following examples of properties capture partial robustness and correctness conditions within and across policies, independently of their implementation. For representing properties, we use a variant of linear-time temporal-logic [20], which expresses conditions on sequences of events. Our safety properties are expressed by formulas using the classical boolean operators $\neg$, $\wedge$, $\vee$, and $\rightarrow$ and by the (past) temporal operators[20]: $\ominus$(previous), $\diamondminus$(once), $\mathcal{S}$ (since), $\mathcal{B}$ (back-to), and (the future operator) $\Box$ (always). Given a finite sequence of events $\sigma = e_1 e_2 \ldots e_n$ and some i where $1 \leq i \leq n$, $\sigma[i]$ denotes the event e_i. Satisfaction of formulas is defined inductively as follows. The formula e, where e is an event, is true at $\sigma[i]$ iff $\sigma[i]$ is the event e. The truth value of boolean formulas is according to the standard interpretation. The formula $\diamondminus p$ denotes whether p was true *once* in the past: that is, $\diamondminus p$ is true at $\sigma[i]$ if and only if (iff) the formula p was true at $\sigma[j]$ for some $j \leq i$. The formula $p\ \mathcal{S}\ q$ denotes whether q happened sometime in the past and that p was true *since* then: it is true at $\sigma[i]$ iff there exists some $j < i$ such that q was true at $\sigma[j]$ and p was true at every $\sigma[k]$ such that $j < k \leq i$. The formula $p\ \mathcal{B}\ q$ is a weak version of $\mathcal{S}$: it is true at $\sigma[i]$ iff either $p\ \mathcal{S}\ q$ is true at $\sigma[i]$ or p has been continually true from the first time instant (that is, p was true at every $\sigma[l]$ for $1 \leq l \leq i$). The formula $\ominus p$ denotes whether the formula p was true at the *previous* time instant: that is, it is true at $\sigma[i]$ (where $i > 1$) iff the formula p was true at $\sigma[i-1]$. The formula $\Box\ p$ denotes whether p is true at every time instant in the sequence. Operator $\Rightarrow$ is defined with $\rightarrow$ and $\Box$: $p \Rightarrow q \equiv\ \Box\ (p \rightarrow q)$.

In the following, we denote an input gate as g_{in} and an output gate as g_{out} as a convention. Operators $\lessdot$ and $\gtrdot$ are used to denote asynchronous *send* and *receive* respectively. Predicate $[g_{out} \lessdot v]$ denotes a sending event, which appends a value v to the queue of gate g_{out}. Receiving event $[g_{in} \gtrdot v]$ checks if the first element in the queue of gate g_{in} matches pattern v. Since we want to specify and check properties capturing how the underlying policy system should behave from the perspective of end users, g_{in} and g_{out} gates are defined in this section with respect to the user interfaces. $g_{in}^{<i>}$ denotes gate g_{in} of the user interface of participant identified by i. Similar is $g_{out}^{<i>}$.

For instance, consider the definition of "available(i)" given in the previous section: "a user is said to be available if and only if the user has not logged out since the last login, and either the screen saver has not gone on since the last time it went off or it has always been off." This statement can be formalized as follows in the temporal logic we consider.

$$\begin{aligned} available(i) \stackrel{def}{=} & (\neg\ [g_{out}^{<i>} \lessdot \text{logout}]\ \mathcal{S}\ [g_{out}^{<i>} \lessdot \text{login}]) \\ & \wedge (\neg\ [g_{out}^{<i>} \lessdot \text{screensaver(on)}] \\ & \quad \mathcal{B}\ [g_{out}^{<i>} \lessdot \text{screensaver(off)}]) \end{aligned} \quad (1)$$

Single-user Properties

The following properties can be evaluated without knowledge of other participants.

1. No module should send or receive messages other than those specified in the description of the architecture.
2. A user should receive a chat invitation only if he or she is one of the intended invitees.

$$\begin{aligned} \forall i : & [g_{in}^{<i>} \gtrdot \text{invite}(SID, Inviter, Invitees)] \\ & \Rightarrow (i \in Invitees) \end{aligned}$$

3. Chat messages should only be received by users who are currently participating in the ongoing session(s).

$$\begin{aligned} & [g_{in} \gtrdot \text{message}(SID, Who, Text)] \\ & \Rightarrow (SID \in S) \end{aligned}$$

   where the set of session ids S is the sessions in which the current user is participating, i.e.

$$\begin{aligned} S = \{SID |\ & \neg\ [g_{out} \lessdot \text{leave}(SID)] \\ & \mathcal{S}\ [g_{out} \lessdot \text{accept}(SID)]\} \end{aligned}$$

Multi-user Properties

This class of properties involve multiple sites.

The first three properties involve accessibility settings of relevant participants. The fourth and fifth properties respectively correspond to reciprocity and non-transitivity. The last property involves message orderings.

1. When user i's door is closed, nobody can invite i for a chat in general, unless exempted by some exception rule specified by i which has not yet been unset.

$$\begin{aligned} & (\neg\ [g_{out}^{<i>} \lessdot \text{set(door(open))}] \\ & \quad \mathcal{B}\ [g_{out}^{<i>} \lessdot \text{set(door(closed))}]) \\ & \wedge [g_{in}^{<i>} \gtrdot \text{invite}(SID, j, Invitees)] \\ & \Rightarrow (\neg\ [g_{out}^{<i>} \lessdot \text{set(door(closed)} \rightarrow \neg\ \text{invite}^{<j>})] \\ & \quad \mathcal{S}\ [g_{out}^{<i>} \lessdot \text{set(door(closed)} \rightarrow \text{invite}^{<j>})]) \end{aligned}$$

2. Everybody can determine whether user i is available except those disallowed explicitly by i.

$$\begin{aligned} & [g_{in}^{<j>} \gtrdot \text{available}(i)] \vee [g_{in}^{<j>} \gtrdot \text{unavailable}(i)] \\ \Rightarrow & \neg\ [g_{out}^{<i>} \lessdot \text{set(true} \rightarrow \neg\ \text{check-availability}(i)^{<j>})] \\ & \mathcal{B}\ [g_{out}^{<i>} \lessdot \text{set(true} \rightarrow \text{check-availability}(i)^{<j>})] \end{aligned}$$

3. No user can check another users' private data unless explicitly allowed. For example, if user i wants to check the real name of user j, permission must have been explicitly given by user j.

$$
\begin{aligned}
&[g_{in}^{<i>} \geqslant \text{name}(\text{real}(X), \text{pseudo}(j))] \\
\Rightarrow\ &\neg\, [g_{out}^{<j>} \leqslant \text{set}(\text{true} \rightarrow\ \neg\ \text{check-name}(j))^{<i>})] \\
&\mathcal{S}\ [g_{out}^{<j>} \leqslant \text{set}(\text{true} \rightarrow \text{check-name}(j))^{<i>})]
\end{aligned}
$$

4. If A has set his name policy to be mandatory reciprocity, then any user B should only see user A's real name if A is also allowed to see B's real name.

$$
\begin{aligned}
&([g_{in}^{<B>} \geqslant \text{name}(\text{real}(X), \text{pseudo}(A))] \\
&\wedge \diamondminus[g_{out}^{<A>} \leqslant \text{set}(\text{reciprocity}(\text{check-name}))]) \\
\Rightarrow\ &(\neg\, [g_{out}^{<B>} \leqslant \text{set}(\text{true} \rightarrow\ \neg\ \text{check-name}^{<A>})] \\
&\mathcal{S}\ [g_{out}^{<B>} \leqslant \text{set}(\text{true} \rightarrow \text{check-name}^{<A>})])
\end{aligned}
$$

5. Anytime if A is allowed to see B's name but not allowed to see C's name, then A cannot see C's name through checking the list of users chatting with B which includes C.

$$
\begin{aligned}
&(\neg\, [g_{out}^{<A>} \leqslant \text{check-chatters}(B)]\ \mathcal{S} \\
&\diamondminus([g_{out}^{<A>} \leqslant \text{check-chatters}(B)] \\
&\wedge (\neg\, [g_{out}^{<B>} \leqslant \text{set}(\text{true} \rightarrow\ \neg\ \text{check-name}^{<A>})] \\
&\quad \mathcal{S}\ [g_{out}^{<B>} \leqslant \text{set}(\text{true} \rightarrow \text{check-name}^{<A>})]) \\
&\wedge (\neg\, [g_{out}^{<C>} \leqslant \text{set}(\text{true} \rightarrow \text{check-name}^{<A>})] \\
&\quad \mathcal{S}\ [g_{out}^{<C>} \leqslant \text{set}(\text{true} \rightarrow\ \neg\ \text{check-name}^{<A>})])) \\
&\wedge ([g_{in}^{<A>} \geqslant \text{chatters}(SID, SetOfChatters)] \\
&\quad \wedge (C \in SetOfChatters)) \\
&\Rightarrow (C \text{ is pseudo})
\end{aligned}
$$

6. Messages sent by any site i are received by another site j in the same order.[1]

$$
\begin{aligned}
&([g_{in}^{<j>} \geqslant \text{message}(SID, i, Text_v) \\
&\wedge \diamondminus[g_{in}^{<j>} \geqslant \text{message}(SID, i, Text_u)) \\
\Rightarrow\ &\diamondminus([g_{out}^{<i>} \leqslant \text{message}(SID, Text_v) \\
&\quad \wedge \diamondminus[g_{out}^{<i>} \leqslant \text{message}(SID, Text_u))
\end{aligned}
$$

Now that we have described some important properties of the system, we show how to systematically test implementations of this system against these properties.

AUTOMATED VERIFICATION

Presence awareness systems, such as the one described here, are composed of elements that can operate concurrently and communicate with each other. These systems are a particular kind of *concurrent reactive systems*, in which every component is viewed as reactive system (i.e., a system that continuously interacts with its environment). An effective approach for analyzing the correctness of a concurrent reactive system consists of *systematically exploring its state space*. The state space of a concurrent system is a directed graph that represents the combined behavior of all concurrent components in the system. For software systems, state-space exploration tools have traditionally been restricted to the exploration of the state space of an abstract description of the system, specified in a modeling language (e.g., [16, 8, 5]). This requires that the entire application must be modeled in the modeling language: a time-consuming and error-prone task. Furthermore, since these state-space exploration tools check an abstract description of the system, errors in the actual application may be missed or misrepresented.

Recently [11], it has been shown how systematic state-space exploration can be extended to deal directly with "actual" code implementing concurrent reactive software systems, in which processes execute arbitrary code written in any general-purpose programming languages such as C, C++, or Java. VeriSoft is a tool for systematically and efficiently exploring the state spaces of such systems. It thus eliminates the need to build a model of the software application to be analyzed.

Overview of VeriSoft

VeriSoft systematically explores the state space of a concurrent software application by controlling and observing the execution of the actual code of all its components. More precisely, VeriSoft can intercept, suspend and resume the execution of specific operations (system calls) executed by the implementation being tested, such as operations on communication objects (e.g., sending or receiving a message). In this way, VeriSoft can drive the execution of the whole system following many scenarios, each of which being defined as a path in the *state space* of the system. VeriSoft can always guarantee a complete coverage of the state space up to some depth; hence, *all* possible executions of the system up to that depth are guaranteed to be covered. Since VeriSoft can typically generate, execute and evaluate thousands of tests per minute, it can quickly reveal behaviors that are virtually impossible to detect using conventional testing techniques. Also, VeriSoft has complete control over nondeterminism, and can completely reproduce any scenario leading to an error found during the search. More details about the state-space exploration techniques used by VeriSoft are given in [11]. VeriSoft has been applied successfully for analyzing several software products developed in Lucent Technologies, such as telephone-switching applications and implementations of network protocols (e.g., see [12]). VeriSoft is available at `www.bell-labs.com/projects/verisoft`.

Four main classes of errors can be detected by VeriSoft: *deadlocks*, *livelocks*, *divergences* and *assertion violations*. These types of errors are discussed in more detail in [11]. We focus here on assertion violations. Assertions can be specified anywhere in the application code with the special operation "VS_assert".

As with all systematic state-space exploration tools, VeriSoft requires an executable representation of the environment (test driver) in which the system operates, in order to close the system and make it self-executable. For this purpose, a special operation "VS_toss" is available to express a valuable feature

[1] However, this does not imply a stronger property: whether all sites see the same messages from all other sites in the same order. This is equivalent to the distributed consensus problem.

of modeling languages, not found in programming languages: *nondeterminism*. This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is nondeterministic: the execution of VS_toss(n) may yield up to $n+1$ different successor states, corresponding to different values returned by VS_toss.

Analyzing Temporal Logic Properties

VeriSoft does not directly check temporal logic properties. Hence, in in order to check the properties described in the previous section, a mechanism is needed to translate these properties into VS_assert statements at appropriate points in the application. Using well-known techniques [28, 2], every such property can be automatically translated into a finite-state machine which can then be used to monitor the execution of the application and raise an assertion violation exactly when the property has been violated.

Analyzing Presence Awareness Systems using VeriSoft

In order to systematically test a presence awareness system using VeriSoft, the interface between communicating entities (e.g., clients and the server) should be instrumented so that messages between entities can be intercepted by VeriSoft (see above). In general, such an instrumentation is always possible; of course, the (one-time) effort for performing the necessary instrumentation will vary for different communication mechanisms (e.g., TCP/IP, UDP, Java Remote Method Invocation, etc.)

The system must then be "closed" by combining it with a set of executable (logical or real) processes representing users. In implementing these processes, the VS_toss operation is used to model the choices among possible user activities, such as changing presence information, enquiring about other users, communicating with other users, and changing presence awareness policies. To keep the state space tractable, some simple assumptions can be made about the users' behavior: for example, one can assume that login messages and logout messages must alternate (i.e., users cannot logout if they are already logged out), and that users cannot send any messages when they are not currently logged in.

Finally, the property monitors must be integrated with each PA Controller and the PA Server. All the events occurring in our properties correspond to messages sent or received by one of the processes: a user interface, a Presence Awareness Controller, or the Presence Awareness Server. Furthermore, all of the properties are local to one of these processes (note that the "multi-user" properties from the previous section are actually local to the Presence Awareness Server). Thus, it is sufficient to have each process check its set of properties every time it either sends or receives a message. This integration can be done by simply extending the message interception described above, to have the corresponding message sent to the property monitors. The property monitors then automatically change their internal state. If a property is violated, the corresponding monitor will automatically execute a VS_assert statement and VeriSoft will report the sequence of operations leading to the error. Thus, checking these correctness properties can be done without changing the internals of the Presence Awareness System.

Checking Properties at Run-Time

Systematic state-space exploration using VeriSoft is done prior to system deployment, by systematically driving the execution of the system through all possible scenarios up to a specific length (depth in the state space). However, our framework also supports a form of testing that can be performed while the actual application is running: the safety property monitors can be left in the system code, since they do not affect the internals of the system. A flag is dynamically set to activate or deactivate the monitors while the collaborative system is running. The monitors, when active, keep track of the (relevant) sequence of events being generated by actual users of the system. If the corresponding property is violated, the monitor reports a violation to the user or (perhaps only) the system administrator. This form of run-time property checking further increases confidence in the correctness of the system.

OUR FEASIBILITY EVALUATION

COCA Implementation of Presence Awareness System

For evaluation, we used COCA [18], a framework in which collaborations are activities involving a group of participants, in which each participant plays a role. A set of coordination rules models each role, reflecting the capabilities and preferences of the participants in the collaboration. The coordination rules are specified modularly in COCA. COCA allows run-time changes to the roles in a collaboration, including modifications to the associated policies.

A COCA virtual machine (*cocavm*) runs at each participant site to enforce the coordination policies by controlling the interactions between this participant and others. A *cocavm* consists of an inference engine and an internal database. The inference engine monitors messages communicated in the collaboration, firing the active rules which unify with each message, and performing actions according to the policy specification. The internal database provides an associate memory for capturing and recording state information regarding the ongoing collaboration.

In our COCA implementation of the presence awareness system, the PA Server/Database and the PA Controllers each are implemented as a *cocavm*. Policies are specified by the user in the COCA specification language [19], in the form of inference rules. The inference engine in the *cocavm* is part of the PA Controller. Upon receipt of a message from either the local user interface or from remote sites, the *cocavm* evaluates the inference rules for a match. The corresponding inference rule is then fired.

The Presence Awareness Controller and the Presence Awareness Server of this implementation of the presence awareness system consist of about 400 lines of policy specification in the COCA specification language. The simple user interfaces consist of about 2000 lines of Java code. The separation between user interfaces and policies supported by COCA makes these interfaces very easy to implement with off-the-shelf design tools. The runtime system of COCA is implemented in Java; the virtual machine plus a few runtime user interfaces sum up to about 20,000 lines of code.

Analysis using VeriSoft

In order to analyze our COCA implementation of the presence awareness system using VeriSoft, we needed to perform the following steps:

- Instrument the COCA environment so that VeriSoft can intercept communication among concurrent entities (i.e. the PA Controllers and the PA Server).
- Automatically translate the properties described earlier into executable monitors. For the translation into Java, we used the translation facility of Triveni [6].
- "Close" the system by writing executable processes corresponding to users. These processes use VS_toss to simulate nondeterminism among the various activities that can be performed by users.
- Limit the number of user processes, in order to keep the analysis tractable. In our analysis, we used either two or three user processes, depending on the properties. In addition, we limited the range of VS_toss in the user processes, to prevent the user process from sending out messages that are irrelevant to the property currently under analysis.

We now describe an example of an important and subtle property violation that VeriSoft automatically discovered, corresponding to the first multi-user property described earlier:

> When user i's door is closed, nobody can invite i for a chat in general, unless exempted by some exception rule specified by i which has not yet been unset.

In this analysis, there are two user processes, user A and user B. User A has a choice among closing his door (all doors are initially open), sending a chat invitation to user B, or setting a policy that user A will not receive a chat invitation from user B when user A's door is closed (initially, no such policies are set). User B's choices are analogous.[2]

After running for 1 hour and 20 minutes on a Sparc Ultra-30 machine with 384MB RAM, VeriSoft automatically discovered the following scenario violating the above property:

1. User A sets his policy prohibiting user B from inviting him for a chat when his (A's) door is closed. This message is sent to the PA Server by A's PA Controller.
2. The PA Server receives this message from A's PA Controller.
3. User B indicates that he wants to invite user A to chat. This message is sent to the PA Server by B's PA Controller.
4. The PA Server receives this message from B's PA Controller.
5. User A closes his door. This message is sent to the PA Server by A's PA Controller.
6. PA Server relays user B's chat invitation to user A's PA Controller.
7. The property monitor at A's PA Controller triggers an assertion violation since A's door is closed and A received a chat invitation from B.
8. The PA Server receives the door close message from A's PA Controller.

In step 6, user A has already shut his door but the PA Server has not yet received it; hence the PA Server relays user B's chat invitation to user A. However, from user A's point of view, he has already closed his door and receives B's chat invitation anyway! This is a property violation since it violates the policy he just set earlier. Indeed this violation is caught by the property monitor, which triggers a VS_assert.

The cause of this property violation is a race condition between A's PA Controller and the PA Server. In particular, there is a delay between the sending of the door-close message by user A and the receipt of the message by the PA Server; in this period of time, there is an inconsistency between user A's actual state and the PA Server's understanding of user A's state. This inconsistency can manifest itself, as in the above scenario, into a property violation observed by the user.

We make the following observations about this error:

- This error results in user A believing that the presence awareness system is flawed. This may lead him to believe that it may also reveal sensitive information about him to other users.
- It is unlikely that this error would have been caught by traditional testing techniques, since there are tens of thousands of possible sequences of user interactions with the system. Moreover, the occurrence of this race condition depends on specific timing conditions that may not be exhibited by a specific testing environment (machine, network, ...), hence making the detection of this error impossible with traditional testing in that environment. In contrast, VeriSoft systematically analyzes the entire state space of the system up to some depth specified by the tester. It is thus guaranteed to find this error eventually since it has complete control over nondeterminism in the system.

In fact, the error is an important flaw in our architecture of the presence awareness system: all policy-related validation is done only by the PA Server since all presence awareness data is stored only in the centralized PA Database. However, since only the PA Controllers have the most recent information about the user's activities, they must also perform some policy-related validation before relaying incoming messages from the PA Server to the user interfaces. Effective validation against these policies may require the PA Controller to have more information about the history of the user's actions: for example, A's policy may say that user B cannot send user A a chat invitation if user A's door is closed and A and B have already been together in a chat session today. Since all presence awareness data is stored in a centralized server in our architecture, the PA Controllers cannot effectively access such history of the user's actions. In order to prevent such

[2] Note that the other choices – opening the door and explicitly allow the other user to chat when the door is closed – are not relevant toward violating the property. That is, for any violating scenario involving these additional actions, there is a shorter scenario with these actions removed that also violates the property.

policy violations arising from message delays, the presence awareness server should actually be distributed across the system (in a consistent fashion).

Note that the flaw discussed above is not specific to our COCA implementation of the presence awareness system, but is a defect in the underlying architecture. Hence, it would likely manifest itself in other implementations of this system architecture, built using specification and development platforms for collaborative applications other than COCA.

CONCLUSIONS AND FUTURE WORK

Presence awareness systems face the following key challenges:

- Policies regarding access to data in collaborative applications have subtle complexities, and must be easily modifiable during a collaboration.
- Users must be able to have a high degree of confidence that the implementations of these policies are correct.

In this paper, we presented a framework to ensure that presence awareness systems conform to complex policies about data access. Our approach is applicable to a wide variety of specification and development platforms for collaborative applications, including several that have appeared in the CSCW literature [4, 9, 25]. We demonstrated the use of our approach by analyzing a presence awareness application implemented in COCA [18], and illustrated the strength of our framework by describing the type of subtle and important errors that can compromise such complex systems.

We are planning to use our approach on a new generation of call processing services for a family of Lucent Technologies switching products. These call processing services involve collaboration between users over a variety of interfaces including the web and the telephone. In particular, users will have access to sensitive presence awareness information, potentially including telephone records, which must not be inadvertently revealed to other users who are not permitted to see it. Users may also want to limit the availability of web-related presence information. The policies governing the access to this presence awareness data are complex, and may be modified during a collaboration (e.g. a user provides more information to another user after verifying his/her identity).

In order to bring this work to bear on the design of this and other complex collaborative applications, several additional issues should be addressed.

How can users specify properties? Various frameworks for awareness notification services provide means for specifying security policies. In COCA, for example, they were specified as inference rules, associated with roles. Most framework developers, we believe, support specification of policies in ways that are fairly intuitive for users. However, the temporal logic we used is perhaps less intuitive for most users to express *properties* that capture partial robustness and correctness conditions within and across policies, independently of their implementation. Since one of the primary issues is giving users confidence that their desired policies are in fact being enforced by the software, it is important that users be able to construct and understand these properties. There are many possible ways of approaching this issue, and future user interface research should explore their effectiveness.

User roles. In this research, we assumed that all users are peers, and all occupied comparable roles in the collaborative work. Future work must address asymmetric situations where, for example, a more powerful class of users have higher priority or more access to certain services or data than other classes of users. These roles may also change on the fly, and policies must address the question of what kinds of privileges are tied to roles, even when the person in that role changes.

More subtle forms of presence information. There may be types of presence information that cannot be controlled to the users' satisfaction with the types of expressions we have looked at in this paper. For example, with an "active badge," I may not care that anyone knows where I am, but I would rather not reveal that I was in the same room as X and Y for two hours. Policies may depend on what can be inferred from multiple sources of data. Also, some forms of communication (e.g., video) have inherent presence information (e.g., I'm at home in my pajamas) and I would like my policies not to be violated indirectly in this manner. Such cases present a much more difficult problem in property specification.

REFERENCES

1. G. D. Abowd. Software engineering issues for ubiquitous computing. In *International Conference on Software Engineering*, 1999.

2. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

3. V. Bellotti, A. Blandford, D. Duke, A. MacLean, J. May, , and L. Nigay. Interpersonal access control in computer-mediated communications: A systematic analysis of the design space. *Human-Computer Interaction*, 11:357–432, 1996.

4. A. Bullock and S. Benford. An access control framework for multi-user collaborative environments. In *Proceedings GROUP '99*, pages 140–149, Phoenix, AZ, November 1999.

5. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

6. C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Objects and concurrency in Triveni: A telecommunication case study in Java. In *4th USENIX Conference on Object Oriented Technologies and Systems*, April 1998.

7. W. K. Edwards. Policies and roles in collaborative applications. In *Proc. of ACM Conf. on CSCW*, 1996.

8. J. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *International Conference on Software Engineering*, May 1992.

9. L. Fuchs. Area: A cross-application notification service for groupware. In *Proceedings of the Sixth European Conference on Computer-supported Cooperative Work*, pages 61–80, Copenhagen, Denmark, September 1999.

10. B. Gaver, T. Moran, A. MacLean, L. Levstrand, P. Dourish, K. Carter, , and B. Buxton. Realizing a video environment: Europarc's rave system. In *Conference on Human Factors in Computing Systems*, 1992.

11. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

12. P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, March 1998.

13. J. Grudin. Why cscw applications fail: Problems in the design and evaluation of organizational interfaces. In *Conference on Computer-Supported Cooperative Work CSCW '88*, 1988.

14. J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, pages 63–70, Sept/Oct 1999.

15. D. Hindus, M. S. Ackerman, S. Mainwaring, and B. Starr. Thunderwire: A field study of an audio-only media space. In *Computer Supported Cooperative Work*, 1996.

16. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

17. S. E. Hudson and I. Smith. Techniques for addressing fundamental privacy and disruption tradeoffs in awareness support systems. In *Computer Supported Cooperative Work*, 1996.

18. D. Li and R. R. Muntz. Coca: Collaborative objects coordination architecture. In *Proceedings of ACM CSCW*, Nov. 1998.

19. D. Li and R. R. Muntz. A collaboration specification language. In *Proceedings of the 2nd USENIX Conference on Domain Specific Languages*, Oct 1999.

20. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer-Verlag, 1992.

21. M. Mantei, R. Baecker, A. Sellen, W. Buxton, T. Milligan, , and B. Wellman. Experience in the use of a media space. In *Proceedings of the CHI'91 Conference on Human Factors in Computing Systems*, pages 203–208, 1991.

22. S. E. McDaniel, G. M. Olson, and J. C. Magee. Identifying and analyzing multiple threads in computer-mediated and face-to-face conversations. In *Computer Supported Cooperative Work*, 1996.

23. T. Nomura, K. Hayashi, T. Hazama, and S. Gudmundson. Interlocus: Workspace configuration mechanisms for activity awareness. In *Computer Supported Cooperative Work*, 1998.

24. L. Palen. Social, individual, and technological issues for groupware calendar systems. In *CHI'99*, 1999.

25. W. Prinz. Nessie: An awareness environment for cooperative settings. In *European Conference on Computer Supported Cooperative Work*, 1999.

26. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, , and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

27. H. Shen and P. Dewan. Access control for collaborative systems. In *Proc. of ACM Conf. on CSCW*, 1992.

28. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *First Symposium on Logic in Computer Science*, pages 322–331, June 1986.

29. R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.

30. E. Zaret. Upstart in the instant messager war. MSNBC, 1999.

31. Q. A. Zhao and J. T. Stasko. Evaluating image filtering based techniques in media space applications. In *Proceedings of ACM CSCW*, Nov 1998.