Triggers: A Dynamic Approach to Testing Multiuser Software

Robert F. Dugan Jr., Ephraim P. Glinert
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York 12180, U.S.A.
{dugan, glinert}@cs.rpi.edu

ABSTRACT
In this paper we describe *triggers*, a novel technique for testing multiuser applications. Triggers provide a framework in which selected application events induce the execution of virtual user test scripts. Unlike conventional execution based verification systems which require complete prescription of multiuser tests, our framework allows the tester to interact with virtual user scripts *in situ*. Additionally, the extensible framework supports any application event or sequence of events and provides an editor to configure the specifics of a triggering event at runtime. Triggers can be used to investigate a variety of problems common to multiuser applications including: synchronization, performance, human-computer interaction, and human-human interaction. We evaluated our framework on a conventionally tested, mature multiuser application. Our evaluation found triggers particularly effective at uncovering race conditions and problems with response time under load.

Keywords
Multiuser Software, Computer Supported Cooperative Work, CSCW, Testing, Verification, Distributed Computing

1 INTRODUCTION

A critical component is missing from multiuser Computer Supported Cooperative Work (CSCW) application development that is taken for granted in single user applications: support for live user testing. Anytime someone wants to test a single user application, they can pose as the user and run the application. It is difficult for a single person to perform a live user test when multiple users are required [4]. State of the art commercial and research testing systems do not provide adequate guidance or support for a single person to perform live multiuser verification. Existing methodologies take a broad based approach to the evaluation of a CSCW application. While acknowledging that technology plays a role in a CSCW system, these methods give few details on how its evaluation should proceed.

Our research has focused on improvements to execution based testing of multiuser or CSCW software. We have developed CAMELOT [4], a technology-focused methodology for testing CSCW software. Developers, user interface specialists, performance engineers and quality assurance personnel can use CAMELOT to evaluate software technology that comprises a CSCW application. CAMELOT provides an organized set of specific techniques that can be used for technological evaluation. The methodology breaks the testing process into two stages: single user and multi-user. In the single user stage, General Computing and Human Computer Interaction features are examined. During the multiuser stage, Distributed Computing and Human-Human Interaction aspects are investigated. A unique code is associated with each technique. The code provides a classification scheme for the tests used and problems uncovered during application evaluation. We believe CAMELOT's techniques are inclusive of most of the technology tests an evaluator would want to perform on a CSCW application.

We devised Rebecca [4], an architecture for an execution based test system, motivated by the desire to support live user participation in a CSCW test. The trigger subsystem described in this paper plays a key role in this area. Test triggers are similar in concept to database triggers. A database trigger executes a set of instructions whenever a predefined condition is met. For example, a database trigger might recalculate the weekly paycheck amount whenever an employee's yearly salary is changed. Rebecca's trigger subsystem monitors each CSCW user. When a user performs a predefined action or sequence of actions, the trigger fires causing the execution of test scripts by one or more virtual users. For example, if a user types "Hello" in a chat application, a test trigger might execute a script causing a different user to respond with "Hi! How are you?"

Triggers radically change the concept of a test session by giving virtual users the ability to react to live users. Instead of being a prescribed process, multiuser testing becomes a dynamic one. Triggers can be used to investigate a variety of problems common to multiuser applications including: synchronization, performance, human-computer interaction, and human-human interaction. These problems are difficult to manually test with limited resources, and challenging or

impossible to test using existing execution based testing systems.

To determine the efficacy of our work, CAMELOT and a Java based implementation of Rebecca were used to evaluate a mature CSCW application. The evaluation uncovered over twenty bugs in the application and provided valuable feedback for future research.

2 RELATED WORK

Execution-based testing focuses on software evaluation in later stages of the software life cycle. Prior work on execution-based testing systems for applications with a strong user interface component can be grouped into three main categories: commercial, academic, and Computer Supported Cooperative Work.

Commercial systems like Platinum Technology's Final Exam C/S-Test [12], Mercury Interactive's Test Suite [9], Rational Software's SQA Suite [14], and JavaStar [19] provide a complete software test environment that includes an architecture for: test execution, test development, test failure analysis, test measurement, test management, and test planning [15]. Recording user activity in a script creates a test. The recording can be played back as a "virtual user" to execute the test. Multiuser testing is supported by the ability to execute several single user scripts in parallel. Virtual users are coordinated using a combination of script synchronization primitives and script execution control provided by the test system's session manager.

Academic research into execution based testing of GUI software has focused on specific problems like script reusability [7, 11], test case automation [1, **?**], script analysis [10], visual program testing [18], and multi-modal record/playback [6, 3]. MITRE's Multi-Modal Logger (MML) improves multiuser testing by allowing the actions of multiple users to be recorded and played back in a single script [3]. The recording automatically imbeds multiuser coordination in the test script.

With the exception of MML, CSCW testing has focused primarily on evaluation methodologies rather than the development of testing architectures. The methodologies take a broad based view of CSCW software. Early work in CSCW evaluation had a strong psychological component. Researchers focused on the group dynamics generated by the introduction of collaborative software into an organization [13]. CSCW specific methodologies include PETRA [17], SESL [13], and MITRE's Evaluation Working Group Methodology [3, 2]. These techniques integrate both the psychological and technological aspects of a CSCW system into the evaluation.

3 ARCHITECTURE

Rebecca is an execution based test system architecture motivated by our desire to support interaction between live and virtual users during application tests. Our discussion of triggers begins with an overview of Rebecca's architecture depicted in Figure 1.

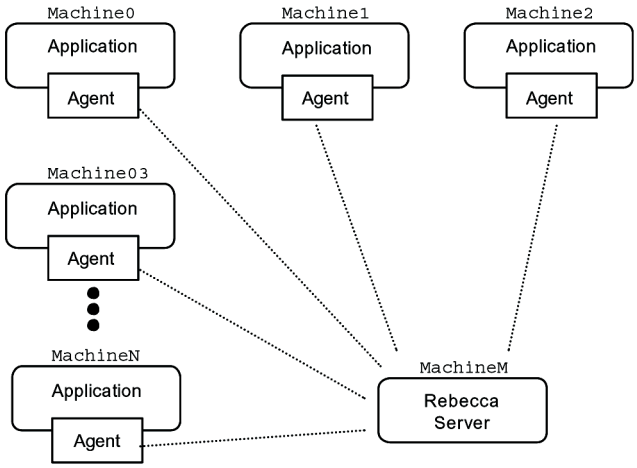

Figure 1: Rebecca's General Architecture

General Architecture

Rebecca's centralized server coordinates one or more distributed testing agents. The server provides a user interface that allows the tester manipulate agents, test scripts and triggers. The server is also responsible for synchronization management that allows coordinated execution of test scripts across agents.

A Rebecca agent is activated for each CSCW user at startup. This is accomplished by adding a small amount of instrumentation to the CSCW application being tested (e.g. two lines of code in the Java-based implementation, Rebecca-J). The agent resides on the CSCW user's local machine. The agent is responsible for responding to server commands, and monitoring local application state.

Rebecca's server commands to the agent include requests to create, edit, save, load, execute, and synchronize test scripts. For remote manipulation of an agent's test script, the server displays a VCR-like control panel and editing window. All editing and control commands are forwarded to the agent that owns the test script. Agent feedback about the state of the test script is sent to the server and displayed in the control panel.

The agent is also responsible for monitoring application state in the form of components and component activity. Rebecca creates virtual user behavior by feeding test script actions into application components. Test script actions are created by recording component activity.

Rebecca provides a flexible, object-oriented mechanism for defining any portion of the CSCW application as a component and any activity in a component as an event. By default, where the implementation language permits, all UI components are automatically defined. Rebecca-J, for example, au-

tomatically monitors all AWT and Swing components and activity that occurs within those components.

Rebecca allows any application activity to be recorded and replayed. A filtering system allows the tester to record specific application events. Integration with existing software development environment is encouraged using the application's programming language as the test script language. Test scripts are reusable as the software evolves through runtime resolution of components and component-centric events. A VCR-like user interface simplifies common record/playback tasks. In addition to triggers, improvements to multiuser test script synchronization include an orchestration metaphor, simplified synchronization mechanisms, deadlock detection, and deadlock recovery.

Trigger Architecture

Rebecca's trigger subsystem executes test scripts for one or more CSCW users in response to component activity. To support trigger management, Rebecca's server provides a user interface to create, edit, load, save, and activate triggers.

Trigger creation involves interaction between the server and one or more agents. Using the server's user interface, the tester selects a triggering agent, application component and component event that will fire the trigger. The tester then selects a test script and executing agent in response to the trigger. The triggering agent and executing agent can be attached to different CSCW users.

Figure 2 shows the trigger management architecture diagram for Rebecca. A detailed discussion of the architecture is oriented around trigger creation and activation. For examples of how to use triggers see Section 4.

In step one, the triggering agent is selected from a list of registered agents. One agent is added to this list for each CSCW application user. As users enter and exit the application, this list is updated automatically. The selection creates a new trigger listener within the agent. The server is given a remote handle to communicate with the trigger listener.

In step two, the tester selects an application component to listen to. The server uses an MVC proxy design pattern to present a remote component browser [5]. Using the browser, the user selects the agent component that will generate the triggering event. If the agent component model changes while browsing, the server will be notified via the proxy model. Because the agent and server execute in different process spaces, the server retrieves the persistent store id of the selected component, rather than its runtime id. The server then passes the component id to the trigger listener using the listener's remote handle.

In step three, a filter for the triggering event is selected. The filter is called a *threshold model*. The threshold model allows the tester to specify precisely the characteristics of a triggering event. For example, the triggering event might be mouse activity inside a GUI push button. A threshold model for mouse activity might be used to select a specific type of mouse activity, such as double clicking the mouse button.

Threshold models register themselves with the server at initialization time. If necessary, the tester can add custom threshold models to the server. Custom threshold models must also register with the server at initialization time. Both the server and the agent are informed of the model selection. On the agent side, default parameters for the model are set. These default values are used to filter component events if the tester chooses not to edit the threshold model. Additionally, the threshold model registers interest with the component that will fire the trigger. On the server side, an editor for the threshold model is configured.

In step four the tester edits the threshold model using the model's editor. The editor allows the tester to customize the threshold for a specific event. Editors are model specific. The editor can be as simple as a set of editable fields describing an event or as sophisticated as a graphical subsystem that specifies the region of the application user interface where event must occur. The result of the editing process is a set of parameters that determine how the threshold will filter component events. These parameters are sent from the server to the threshold model residing in the agent.

In step five, the script executing agent is selected from a list of registered agents. The tester's selection creates a new recording player within the executing agent. The trigger agent is given a remote handle to communicate with the executing agent.

In step six, the trigger agent's CSCW user begins using the application. The user can be either a live or a virtual user executing a test script. The trigger agent threshold model filters events generated by the tester selected component. If an event meets the threshold condition, the trigger fires.

Returning to the mouse event example, when the cursor is inside the push button region, all mouse events are sent to the threshold model. The model ignores the events until the user presses the mouse button generating a double click event. Once received, the mouse press event causes the trigger to fire.

In step seven, the trigger is finally fired resulting in two actions. First, the trigger agent activates the execution agent's recording player. This will cause the player to replay its recording. Second, the server side trigger counter is incremented. The change is displayed in a counter field associated with the trigger on the server. A check is performed to see if the firing count is exceeded. If exceeded, the trigger agent is disabled so that no more firings can occur.

Threshold Models

Selecting a threshold model is an important part of the trigger process. Events generated by the selected component are passed to the threshold model for filtering. If the event or sequence of events passes the threshold filter, then the trigger

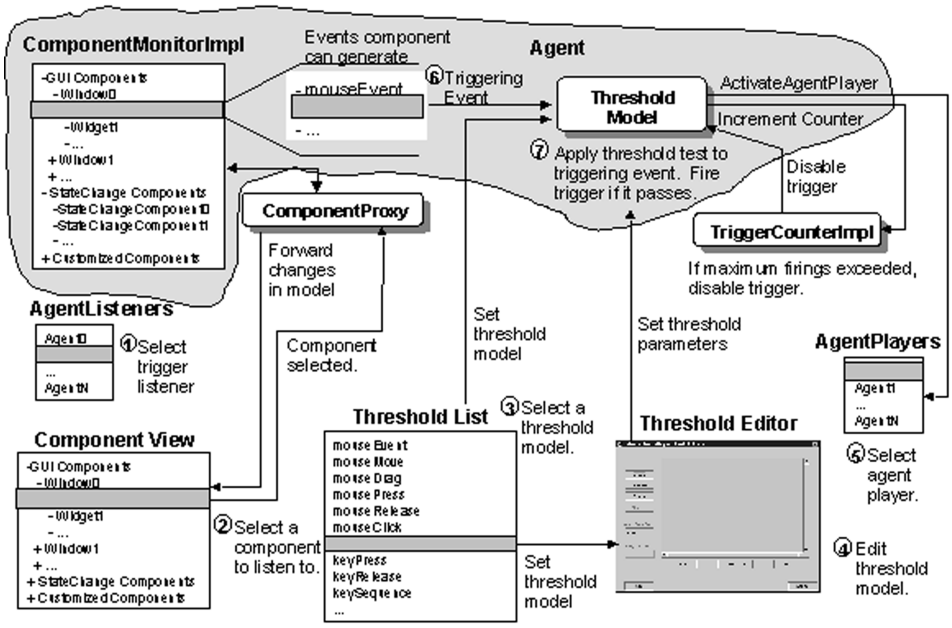

Figure 2: Rebecca's Trigger Architecture

fires. The exact nature of the filter depends on the threshold model selected. It can be as simple as ensuring that the event is of a specific type like a mouse event in a push button, or as complicated as a state machine that reaches its end state with the correct sequence of events like keyboard strokes in a text area producing the word "Hello". Rebecca-J implements a number of threshold models (see Table 1).

Threshold Model	**Description**
`keyPressed`	`KEY_PRESSED` event
`keyReleased`	`KEY_RELEASED` event
`keyTyped`	`KEY_TYPED` event
`mouseClicked`	`MOUSE_CLICKED` event
`mouseEntered`	`MOUSE_ENTERED` event
`mouseExited`	`MOUSE_EXITED` event
`mousePressed`	`MOUSE_PRESSED` event
`mouseReleased`	`MOUSE_RELEASED` event
`mouseMoved`	`MOUSE_MOVED` event
`mouseDragged`	`MOUSE_DRAGGED` event
`propertyChange`	state change component generates a `PROPERTY_CHANGE` event
`keySequence`	sequence of `KEY_PRESSED` events
`mouseRegion`	mouse event in designated GUI location

Table 1: Threshold Models implemented in Rebecca-J

Depending on the threshold model, an editor may be available for runtime customization. The editor may be as simple as a set of form fields specifying characteristics of the event or as sophisticated as a graphical editor indicating the area of a GUI the event must occur.

`mousePressed` *Threshold Model*
Rebecca-J's mousePressed model typifies a simple event type threshold model. The threshold model examines all mouse events generated by the selected component. If the mouse event is of type `MOUSE_PRESSED`, then the trigger fires. Because of the simplicity of this threshold model, no editor is necessary.

`mouseRegion` *Threshold Model*
Rebecca-J's `mouseRegion` model is an example of a sophisticated threshold model. An editor, shown in Figure 3 provides runtime properties of the mouse event that must be satisfied for the trigger to fire.

After the tester has used the remote component browser to select a GUI component, the `mouseRegion` threshold editor is activated. The editor queries the remote component for its dimensions and displays a facsimile in a graphical editing window. The tester selects one or more geometric regions and draws them on top of the facsimile. Finally, a specific type of mouse event is selected from a pull-down list. Only events that occur in the tester drawn regions with the specified type will fire the trigger.

Figure 3 gives an example of the `mouseRegion` editor in use with an application. The `JButton1` button component is selected using the remote component browser. The `mouseRegion` editor gets the dimensions of the button remotely and displays them in the graphical editing window. The tester presses the `Zoom In` button several times to enlarge the push button facsimile. The rectangle geometric region is selected. The tester draws a rectangle in the upper right quarter of the facsimile. The tester then selects the

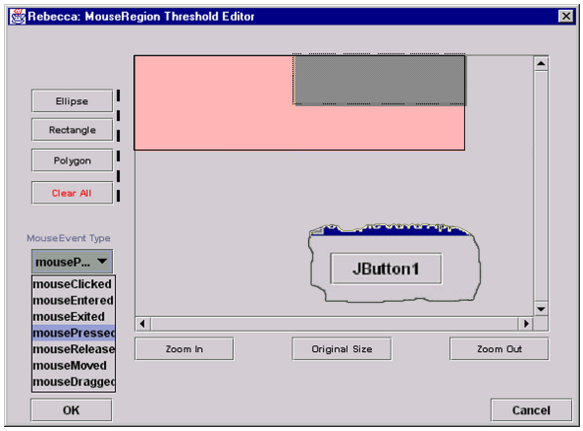

Figure 3: `mouseRegion` threshold model editor

`mousePressed` event from the pull-down list and presses the OK button to end the editing session. The threshold model is now configured to fire whenever the CSCW user presses the mouse button in the upper right corner of the `JButton1` push button.

Threshold Model Customization

Like most of Rebecca's subsystems, the threshold model subsystem is extensible. A tester can create new threshold models that weren't anticipated at the time Rebecca was implemented. Threshold model customization requires three steps: model registration, comparator definition, and editor definition. The model must be registered with the system so the user can select it from the server's trigger panel user interface. The comparator determines if an event received by the trigger listener from the selected component is equivalent to an event stored in the threshold model. The editor allows runtime configuration of the threshold model.

Rebecca uses the MVC design pattern for threshold model editors. The model is the threshold model itself. The threshold view displays representation of the model and the model's configuration parameters. The threshold controller converts user actions directed at the view into commands that manipulate the model.

Event Sequence Triggers

In addition to firing a trigger in response to a single event, Rebecca provides the ability to fire after a sequence of events. For example, instead of firing whenever a keyboard event is generated, Rebecca can fire when a specific sequence of keyboard characters is typed (e.g. "Hello"). A runtime configurable event state machine to supports this capability. Events generated from the selected component are ignored until one equivalent to the first input condition event in the state machine is generated. The machine progresses through states until it reaches a final state or until an event occurs that does not satisfy a move to the next state. When the machine reaches its final state, the trigger is fired and it returns to the start state. Rebecca-J implements this state machine in the abstract class `ThresholdModel` method `isThresholdMet()`.

One Agent, Multiple Triggers

In some situations, a single threshold model is not sufficient to express the conditions under which a trigger should fire. For example, consider a trigger for a `PropertyChangeInt` event that fires if the new state is less than ten or greater than twenty. The `PropertyChangeInt` threshold editor allows only one boolean condition to be specified on the event. Rebecca allows the user to create multiple triggers on the same component and event in the same agent. This allows one trigger to be created for each boolean condition. In the state trigger example, one trigger would be created for values less than ten and a second for values greater than twenty. Both triggers would exist in the same agent, listen for the same events from the same state component, and replay the same recording.

Multiple triggers allow the user to OR a set of threshold model tests on an event, but what about other boolean operations? Rebecca does not provide for any other boolean operation. In order to provide complete a complete set of boolean operations, a boolean algebra and would need to be adopted. The operators in this algebra would consist of AND, OR, and NOT. The variables in the algebra would be a list of trigger names. Triggers specified in a boolean expression would never fire. A boolean expression would be assigned its own trigger. If the boolean expression was satisfied, then its trigger would fire. Consider an example where two state triggers, `TriggerGreaterThan10` and `TriggerLessThan20` are set on the same `PropertyChangeInt` component. `TriggerGreaterThan10` fires when the new integer value is greater than ten. `TriggerLessThan20` fires when the new value is less than twenty. The syntax for a boolean expression trigger that fired for integer value changes between ten and twenty would look like: (`TriggerGreaterThanl0 AND TriggerLessThan20`).

Timers

There are some situations where a virtual user script may need to be activated by a timer, rather than application activity. For example, the tester may want virtual user to manipulate a widget or type some characters every few seconds. Periodic load like this is useful for observing performance characteristics of the application under test. Rebecca's trigger architecture includes provisions for such time-based triggers.

Unlike other triggers, timers are managed entirely by the server so it is not necessary to select a remote agent component and threshold model. The tester configures the timer by double clicking on the timer widget that appears with every trigger panel in the server. A timer selection window appears and is used to manage timers once they have been configured. The tester can add a new timer, delete existing

timers, edit an existing timer, or attach the selected timer to the trigger panel.

Timer triggers are managed by the server, rather than by remote agents. This allows multiple recording players to be attached to the same timer. When the timer fires, all of the recording players are activated near-simultaneously. Simultaneous replay of recordings creates realistic approximations of application use. For example, periodic dialog between a group of users in a chat application could be simulated.

True simultaneous replay of recordings, implemented with networking techniques such as broadcasting, makes for unpredictable testing. The tester can never be sure if the replay started at exactly the same moment on each machine. Software and hardware layers throughout the command's path from server to agent introduce delays that can keep it from being processed immediately. The cause of these delays can change during the test session making the replay order different each time the trigger fires. When a timer trigger fires Rebecca iterates through an ordered list of players, activating each in turn. The server provides the tester with a user interface to configure the ordered list.

4 USAGE CASES

Triggers are a powerful mechanism for testing distributed applications. The principal advantage of triggers over traditional test systems is that a live user can be incorporated into a test session. Virtual users react to events generated by other users, live or virtual. This reactive approach creates test sessions that are dynamic, rather than completely prescribed. The next several sections present examples of trigger use.

Usage Case 1: Synchronization Problems

Race conditions are a common synchronization problem in distributed systems. A race condition occurs when system behavior is dependent on instruction ordering between threads of execution. One way to test for the presence of a race condition is to create a situation using parallel threads varying the execution order.

Consider a simple distributed application with two buttons `+/-` and a counter display. Whenever a button is pressed, the counter is incremented/decremented by one and the new value is displayed in the text field of all distributed copies of the application. A potential race condition exists when users press the count buttons near-simultaneously.

Triggers can help test for this race condition. A trigger could be set up between the live user and a virtual user. Whenever the live user presses the `+` button, the virtual user immediately reacts by pressing the `-` button. If the count field reads `0`, then no matter how many times the live user presses the `+` button, it should always read `0`.

The probe for the race condition can be coupled with a stress test by creating a second trigger. Stress tests help uncover application flaws that aren't exposed under normal use. The new trigger presses the `+` button whenever the original virtual user presses the `-` button. Now we have a situation where two virtual users are reacting to each other. The maximum trigger count field in the panel for the new trigger is set to `1000`. The test begins with the live user pressing the `+` button. Assuming the count field begins with `0`, at the end of the test it should read `1`. In addition to using duration to stress test the system, stress can be increased by setting the playback delay on the trigger recordings to `NO_DELAY`.

Testing for race conditions is possible with a traditional testing system. One advantage of prescribed testing over triggers is the ability to test with scripts executing simultaneously on separate machines. In contrast to the simplicity of triggers, however, some effort is required to produce the test case. For the test described in this section, the tester would be forced to create two scripts, one for each virtual user. Some form of synchronization primitive would have to be added so that both scripts began execution at the same time. A looping construct would be necessary so that the button presses could be repeated. Finally, the modified scripts would have to be saved, compiled, and loaded into the test system.

Usage Case 2: Performance Problems

Another vexing difficulty when developing synchronous multiuser applications is investigating performance issues like the responsiveness of the system under load. Response time is particularly important for user interface portions of the application. User anxiety rises when interactive components, such as a slider bar or pull down list, do not respond within milliseconds [19].

Consider another simple distributed application with a push button and slider bar. The tester wants to investigate the performance effect that pushing the button in one copy of the application has on the moving the slider bar. Slider bars are useful way to get a "feel" for the responsiveness of the application. If the cursor doesn't track well with a slider bar, then the user will notice it immediately.

The tester creates a trigger that activates a virtual user that presses the push button. The trigger fires whenever the live user moves the slider bar. During testing the slider bar does move sluggishly indicating a performance problem. Curious about whether part of the problem is due to a backlog of queued push button events, the tester resets the trigger's threshold model to use the mouseRegion model. The left half of the live user's slider bar is marked as the triggering region for the virtual user. Now the tester can compare how the slider bar behaves when inside and outside the triggering region. If the slider bar is sluggish inside the region, but immediately responsive outside it, then the user knows that an event backlog is not the problem.

Compared to triggers, traditional testing systems only have the ability to place a gross load on the system. This is accomplished by running one or more iterating scripts against the application. A live user can participate in such a test session, but only in an uncoordinated fashion with the scripts.

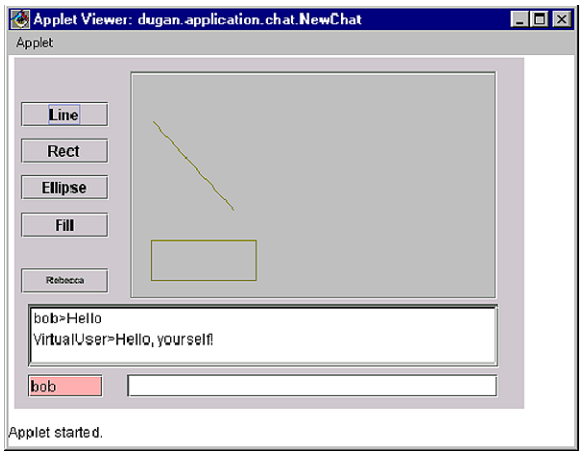

Figure 4: A shared drawing/chat application.

Using the example from this section, the tester executes a script that repeatedly presses the push button in one copy of the application. A live user interacts with the slider bar in another copy of the application while this script executes.

Since live user-based script control is unavailable with traditional testing, all interactions occur while the script executes. Script execution can be controlled from the test system, but it would be difficult or impossible to implement the test cases in this section. When the cursor nears the slider bar, interaction would stop while the tester configured and started the push button script. Interaction would then continue and as the slider bar's behavior was observed under load. When the cursor was about to leave the slider bar, interaction would stop while the tester terminated the push button script. The test cases would be impossible if the application and test system were on the same machine because the cursor would have to leave and re-enter the application to control the test script. The mouseRegion trigger test case would be impossible to imitate because by the time the tester had shut off the test script and returned to the application, the backlog of queued events would have been processed.

Usage Case 3: Simulating User Behavior
In some situations, the tester may desire one or more virtual users to simulate "normal" user behavior. This allows the tester to observe how the application behaves under normal use. The definition of normal is application specific. Generally, the tester assigns a profile to each virtual user consisting of initiated and reactive behaviors. Initiated behavior consists of actions performed by the virtual user without regard to other activity in the application. For example, the virtual user might perform a series of mouse movements culminating in a button press every couple of minutes. Rebecca can model this behavior by replaying a test script continuously, or using a timer trigger.

Reactive behavior consists of virtual user actions triggered by other user activity. For example, when testing a chat program, the virtual user might send the message "Hello, yourself!" in response the string "Hello" being typed by another user. Rebecca models this behavior using triggers that fire based on the activity of other users.

Consider the shared drawing/chat application in Figure 4. The tester wants to create a profile for a virtual user that will interact with the live user during testing. For initiated behavior, the virtual user will draw several shapes every sixty seconds. This behavior provides a periodic load on the system typical of normal use. Using Rebecca, the tester creates a recording that draws several shapes. A timer trigger that fires once every sixty seconds is attached to the recording.

The virtual user also has three reactive behaviors in its profile. The first behavior is a friendly attitude when the live user types "Hi", "Hello", or "Hey" in the chat window. The virtual user responds with "Welcome. I'm drawing in the upper right corner and using blue!" The tester sets up this behavior by recording the welcome message and copying the recording to `keySequence` triggers for each possible live user greeting.

The second behavior is a protective attitude about the upper right hand quarter of the drawing area. Whenever the live user's mouse enters this area and begins drawing, the virtual user sends the angry message "Hey! I'm drawing in this area, draw somewhere else!" This behavior is set up by recording the angry message and attaching a `mouseRegion` trigger to the recording. The `mouseRegion`'s editor is configured to fire to `mousePressed` events in the upper right quarter of the drawing area. The live user's agent is selected as the trigger listener.

The third behavior is another protective attitude towards the drawing color used by live user. The virtual user only draws with the color blue. If the live user chooses blue as well, the virtual user sends the angry message "Hey! Blue is my color. You've ruined the picture and we are starting over again." and clears the drawing area. If the live user chooses a different color, the virtual user sends the message "Great choice! That will complement my blue very nicely." In order to describe this behavior to Rebecca, a new component, event, and threshold model for `propertyChangeColor` must be written.

The specifications for the `propertyChangeColor` are as similar to `propertyChangeInt`. The component is a state change component. It contains the current value of the local user's drawing color. The event is a property change event. It is generated by the component when the user's drawing color changes. The value transmitted in the event is the new drawing color. The threshold model tests for changes to the drawing color component. An editor configures the model by allowing the user to select a drawing color from a list in combination with a "this" or "all-but-this" option. If this is chosen, then the trigger will fire when the drawing color is selected. If all-but-this is chosen, then the trigger will fire when any other color is selected.

The virtual user's color selection behavior is created with two triggers. The first trigger is attached to a recording that sends the angry message and erases the drawing area. A `properyChangeColor` threshold model is configured to fire when the live user selects the color blue. The second recording sends the friendly message. It is attached to a `propertyChangeColor` trigger that fires when the live user selects any color except blue.

Rebecca's support for reactive behavior in virtual users greatly expands the possibilities for simulating user activity. Traditional testing systems do not have this capability. For initiated behavior, Rebecca's continuous replay and trigger timer mechanisms offer a simple alternative to the labor intensive task of adding wait commands and loop constructs, compiling, loading and running a traditional test script.

Usage Case 4:A Flurry of Activity: Trigger Chaining
The usage cases discussed in this section have dealt with virtual users reacting individually to live user actions. Rebecca's trigger subsystem also supports trigger chaining. Trigger chaining allows virtual users to react to each other in a coordinated fashion. A trigger chain is created when a trigger is fired in reaction to event generated from a virtual rather than a live user. Triggers can be strung together across many virtual users to create a chain reaction to a single event.

Consider the following scenario. Three triggers are registered with the live user's agent. Any one of these triggers will cause the same recording in VirtualUser0's agent to execute. Two triggers, TriggerD and TriggerE, registered with VirtualUser0 fire because of an event in this recording. TriggerE causes the replay of a recording in VirtualUser2, where a branch of the chain terminates. TriggerD activates a recording in VirtualUser1 that fires TriggerF. TriggerF replays a recording in VirtualUser3 where the last branch of the chain terminates.

Figure 5 shows how trigger chaining can be used to extend the shared drawing area test from the previous use case. A second virtual user is added to the test. As in the original test, when the live user types a greeting in the chat window, the original virtual user, VirtualUser0, responds with "Hello". This response, however, is chained to another trigger that causes VirtualUser1 to type "Don't be fooled by the friendly greeting. VirtualUser0 is actually very testy. If you want I can prove it to you. Just say 'show me'" in the chat window.

If the live user types any phrase containing "show me" in the chat window, VirtualUser1 replays a recording of mouse movements in the upper right corner of the drawing area. Chained to these mouse events is VirtualUser0's angry response "Hey! I'm drawing in this area, draw somewhere else!" This triggers a retort from VirtualUser1 "See, I told you! I'll tell you something else. Don't even think about drawing in blue. VirtualUser0 really hates that." VirtualUser0 gets in the last word with "I heard that!" triggered by the key sequence "hates that" from VirtualUser1.

Although unimplemented in Rebecca-J, Rebecca has a second form of chaining: trigger state chaining. Trigger state chaining allows the tester to construct a non-deterministic finite automaton from a set of triggers. The input grammar for the automaton is quadruple of component, event, threshold model, and trigger count for each trigger in the chain. State transitions occur when the input condition for the current state is satisfied (the trigger fires).

State change triggers are useful in situations where the tester would like a trigger to replay different recordings as the test session progresses. Consider the VirtualUser0's response when the live user selects the color blue. Instead of sending the message and erasing the graphics in a single recording, the virtual user's response could be broken up into a collection of smaller responses. The first time the live user selects the color blue VirtualUser0 issues a warning: "Hey! Blue is my color, you'll ruin the picture if you use blue too." If the live user selects another color, then the state machine returns to the start-state. If, however, the live user begins to draw with the color blue, VirtualUser0 issues another warning: "Stop drawing in blue or I'll erase the whole picture!" If the live user selects another color, then the state machine returns to the start-state. However, if the user continues to draw VirtualUser0 sends the message "I'm going to keep erasing the drawing area until you change your drawing color from blue." and clears the drawing area. This message/erase recording is triggered by live user drawing activity until another drawing color is selected.

Although Rebecca-J has not implemented trigger state chaining, what would the implementation look like? Each trigger panel would include a state chaining button. When the button is pressed the user would see a dialog box listing all registered triggers. The user selects one or more of the registered triggers and adds them to the next state list. All triggers in the next state list are disabled until the trigger being edited has fired the maximum firing amount of times. The tester can check the "Reset Count" box for triggers in the next state list. This resets the trigger fired count for that trigger to zero when the trigger is enabled. The "Reset Count" box allows movement to a previous state in the trigger state chaining diagram.

5 EVALUATION

Our triggering architecture and implementation was included in an overall evaluation of CAMELOT and Rebecca [4]. The Reconfigurable Collaboration Network (RCN), a mature CSCW shared windowing application was the application under test [16]. RCN allows multiple users to share control of a single public machine using a remote machine with only one user in control of the public at a time. Race condition and response under load triggers were particularly effective.

From early testing, it appeared that the public machine was maintaining state information about the controlling remote machine's keyboard. We were concerned that a race condi-

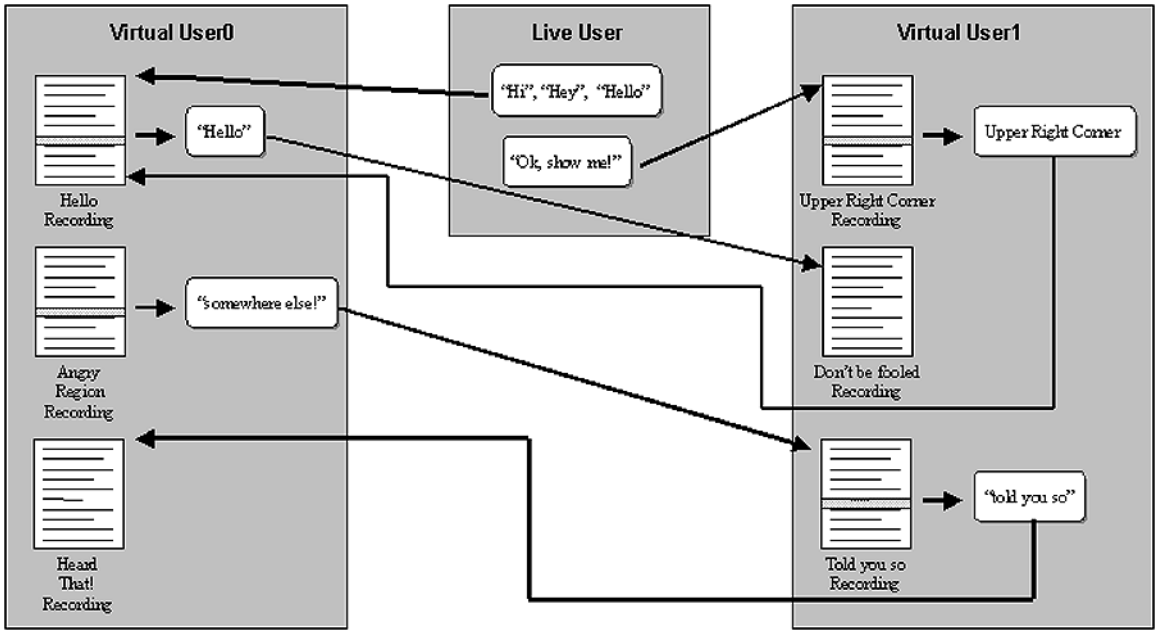

Figure 5: Trigger chaining extends shared drawing area test.

tion might arise when one user took unexpected control of the public machine from another user. To see if this created a race condition, a recording for a virtual user taking control of the public machine and typing "aaa" was made and connected to a trigger. The trigger fired whenever the live user was in control of the public and pressed the SHIFT key. During the tests, the public machine interpreted UserB's keyboard presses as "AAA". Additional trigger test using the ALT and CTRL keys indicated the same problem.

A similar trigger test determined that if the live user did not release the mouse button before the virtual user took over the public, then all virtual user mouse events were interpreted as mouse button press events. The test also discovered that the live user could still control the public machine's screen cursor as long as the mouse button stayed pressed. Only the ability to integrate live and virtual users in a single test session made this discovery possible.

Several other race conditions were exposed using triggers including problems joining sessions, users, teams, and public machines.

We were also able to user triggers and threshold models to explore the response time of the application under load. We were particularly interested in the effect that RCN's ghost cursors would have on the public machine's system cursor and vice-versa. Because all mouse events had to be transmitted across the network to a central server on the public, this centralized design had the potential to cause a performance bottleneck. The mouse region threshold editor was used to create a trigger that would fire when the public's system cursor entered the top half of the screen. The trigger was attached to the recordings of ghost cursor movement for several virtual users. Several tests were conducted using a trigger based on the mouse region threshold editor, the public system cursor, and RCN's ghost cursor. Results from this testing showed the system did not scale well and uncovered memory leaks on both the public and private machines.

6 CONCLUSION

Commercial and academic testing systems that provide multiuser support completely prescribe the test session from start to finish precluding live user participation. Virtual user orchestration is specified using a combination of script synchronization primitives and session manager scheduling.

Triggers radically change the concept of a test session by giving virtual users the ability to react to live users. Instead of being a prescribed process, multiuser testing becomes a dynamic one. Triggers operate on any application activity that can be recognized by the test system. A set of triggers can be defined on the same event or chained together to provide complex virtual user behavior.

Threshold models provide sophisticated control over the firing of a trigger. If an event or sequence of events passes the threshold condition, then the trigger fires. Several models were introduced for user interface events (e.g. `mousePressed`,`mouseRegion`), application state change (e.g. `propertyChangeInt`), and timers. Threshold models range in sophistication from the simple (e.g. a mouse button press) to the complex (a specific type of mouse button press in a specific region of a user interface component) and are completely customizable.

Usage cases were presented to demonstrate the capabilities

of triggers. These included detecting race conditions, determining response time under load, simulating realistic user behavior, and the ability of virtual users to react other virtual users.

An evaluation of triggers was conducted on RCN, a mature shared windowing CSCW application. Serious flaws in RCN were uncovered prompting the following comment when Rebecca uncovered one troublesome race condition:

> `i don't think any tester would have ever discovered that. simply for discovering that, i consider rebecca a success.` *J.J. Johns, RCN Developer*

7 ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under awards CDA-9634485 and CCR-9527151. The authors would also like to thank Professor Edwin H. Rogers and software developer J.J. Johns for their support and advice during the evaluation of RCN using Rebecca-J.

REFERENCES

[1] D. Cohen, S. Dalal, A. Kajla, and G. Patton. The automatic efficient test generator (aetg) system. In *International Conference on Testing Computer Software*, Washington, DC, 1994.

[2] J. Drury, L. Damianos, T. Fandercla, L. Hirschman, J. Kurtz, and B. Oshika. Scenario-based evaluation of loosely-integrated collaborative systems. In *Proceedings of Conference on Human Factors in Computing Systems (CHI '00)*. ACM Press, 2000.

[3] J. Drury, L. Damianos, T. Fanderclai, L. Hirschman, J. Kurtz, and B. Oshika. Methodology for evaluation of collaborative systems, http://www.mitre.org/support/papers/tech_papers99_00/damianos_evaluating/index.shtml, 1999.

[4] R. F. Dugan Jr. *A Testing Methodology and Architecture for Computer Supported Cooperative Work Software*. Doctoral thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1995.

[6] M. L. Hammontree, J. J. Hendrickson, and B. W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of Conference on Human Factors in Computing Systems (CHI '92)*, pages 431–432. ACM Press, 1992.

[7] L. R. Kepple. The black art of gui testing. *Dr. Dobb's Journal*, 19(2):40–42, 1994.

[8] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for guis. In *International Conference on Software Engineering*, pages 257–266, Los Angeles, California, 1999. ACM Press.

[9] MercuryInteractive. Testdirector: Scalable test management. User's guide, Mercury Interactive Corporation, 1997.

[10] H. Okada and T. Asahi. Guitester: A log-based usability testing tool for graphical user interfaces. *IEICE Transactions on Information and Systems*, E82-D(6):1030–1041, 1999.

[11] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for gui systems. In *ACM SIGSOFT Internation Symposium on Software Testing and Analysis*, volume 23, pages 82–92, Clearwater Beach, Florida, 1998. ACM Press.

[12] PlatinumTechnology. Final exam c/s-test. Tutorial, Platinum Technology, Incorporated, 1997.

[13] M. Ramage. *How to Evaluate Cooperative Systems*. Doctoral thesis, Lancaster University, Department of Computing, 1999.

[14] RationalSoftware. Sqa manager user's guide. User's guide, Rational Software Corporation, 1997.

[15] D. Richardson and N. Eickelman. A framework for software test environments. In *Eighteenth International Conference on Software Engineering (ICSE-18)*, 1996.

[16] E. H. Rogers, C. Geisler, J. Farley, J. Johns, and C. Parker. The reconfigurable collaboration network, a demonstration of collaborative system sharing. In *Proceedings of the European Computer Supported Cooperative Work Conference 1999*, Amsterdam, Netherlands, 1999.

[17] S. Ross, M. Ramage, and Y. Rogers. Petra: Participatory evaluation through redesign and analysis. *Interacting with Computers*, 7(4):335–360, 1995.

[18] G. Rothermel, L. Li, C. DuPula, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *20th International Conference on Software Engineering*, pages 198–207, 1998.

[19] SunMicrosystems. Javastar overview, 2000.

[20] L. J. White. Regression testing of gui event interactions. In *International Conference on Software Maintenance*, pages 350–358, Monterey, California, 1996. IEEE Computer Society Press.