UI Traces: Supporting the Maintenance of Interactive Software

Andrew Sutherland
Department of Computer Science
University of Saskatchewan
110 Science Place
Saskatoon, SK, Canada
andrew.sutherland@usask.ca

Kevin Schneider
Department of Computer Science
University of Saskatchewan
110 Science Place
Saskatoon, SK, Canada
kevin.schneider@usask.ca

Abstract

We propose a method to support the maintenance of interactive software systems with user interface traces, that involves: (1) collecting execution traces of an interactive system, (2) segmenting execution traces into user interface traces according to user interface activity, and (3) mapping the user interface activity to the implementation activity.

To support our approach, we developed a tool that uses aspect-oriented programming and load-time weaving to collect user interface traces from an interactive system. The tool allows us to browse the user interface traces and view user interface related data such as: user input, display updates, and thread activity. Using our tool, we demonstrate how developers can orient themselves and identify the slice of code relevant to performing common software maintenance tasks.

1. Introduction

Maintaining interactive systems can involve significant engineering and design challenges. The inclusion of a user interface means that a balance must be maintained between processing user input, performing computations, and providing appropriate feedback and updating the display. We propose that software maintenance tasks on interactive systems benefit from tools that allow user interface activity to be matched to implementation activity. For instance, clicking a button widget is a simple process from the user perspective, but there may be a great deal of computation that occurs as a result. With current tools it is difficult to get an impression of what occurs at the implementation level when such an interface action does occur.

In this work, we discuss bridging the gap between the "user interface perspective" and the "implementation perspective". Our approach records execution activity when specific actions occur in the user interface. We group the execution activity associated with a specific action into a user interface (UI) trace. These UI traces can be used to assist a developer in navigating architectural components and source code.

2. User Interface Traces

Execution traces generated during specific usages of a target application are verbose and it is difficult to isolate the portion of the trace that is of interest. To allow a developer to effectively identify portions of an execution trace that are related to user interface actions, we segment the raw execution traces according to *interface actions*. A *user interface action* (UI action) is defined as a single event or group of events that occurs on the user interface. This includes mouse events (motion, drags, clicks), keyboard events (single characters or blocks of text), and display updates. In an execution trace, UI actions are represented as method calls. Table 1 lists some of the UI actions found in a typical interactive system implemented in Java. Depending on the specific implementation the method invocations may be different as there are multiple ways to capture some of the actions listed.

In an interactive system, UI actions result in or are a result of computation. For instance, mouse clicks that result in a button widget being activated, typically have some effect or cause the state of the application to change. Display updates are preceded by computation responsible for determining the new image or colour to be displayed, and possibly for coordinating the swapping of display buffers. In our approach, we segment the raw execution trace by locating the interface actions, i.e. the corresponding method calls, and grouping the computation that falls in between the interface actions.

We define a UI trace as: (1) the user interface action that initiates the user interface trace (e.g., a mouse press, a key press, a display update, the application launch, or a

Table 1. Common UI actions and corresponding method invocations

UI Action	Corresponding method invocation	Associated Data
Mouse click	`mouseClicked(MouseEvent)`	Widget ID, x, y coordinates
Mouse move	`mouseMoved(MouseEvent)`	x, y coordinates
Key press	`keyTyped(KeyEvent)`	Key code, option keys
Display update	`paint(Graphics)`	Component painted
Cursor changed	`Cursor(String)`	Cursor name
Sound played	`AudioClip.play()`	File played
Thread Signals/Timers	`Thread.run()`, `Thread.sleep()`	Thread id
Application Launch	`main(String[])`	
Application Exit	`exit()`	

user interface related thread event), (2) data associated with the user interface action (e.g., pointer location, modifier key state, thread number), (3) a screenshot of the user interface (or relevant portion of the user interface) when the user interface action occurred, and (4) the execution activity resulting from the user interface action (e.g., the sequence of calls and the resource utilization from the time of the user interface action until the next user interface action).

UI traces support software maintenance tasks in three ways. They segment lengthy execution traces so that they are more manageable. They map user interface activity to the application's implementation. Finally, they provide the software maintainer with a 'slice' of the implementation (e.g., code) that is specically related to the use case being performed.

3. Prototype

To determine whether it is viable to support navigation with UI traces we prototyped a tool that collects UI traces in the manner we have described and provides a visual representation of the architectural components that are active during a UI trace along with a screen shot of the user interface (cf. Figure 1). The source code corresponding to the architectural entities can then be brought up in an editor window with the appropriate sections highlighted.

In our prototype, we use an AspectJ [5] implementation to describe and capture the interface actions that occur in a target application. We have written a tracing aspect that records when any method in the target application is invoked. The tracing aspect also tags method invocations that are associated with UI actions. These tags are used to segment the entire execution trace by the UI actions. The method invocations associated with a UI action can vary depending on the application, however, the second column in Table 1 lists the methods usually associated with the more common UI actions.

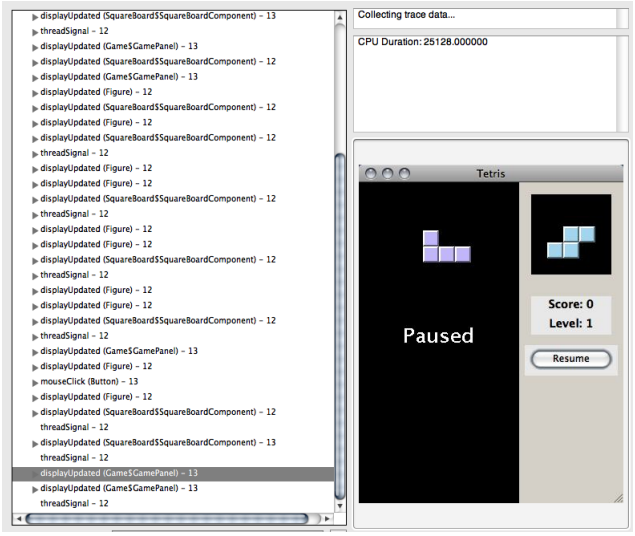

Figure 1. Main interface of the prototype. Top left panel: UI trace list. Bottom right panel: recorded screen grab. Top right panel: activity and performance data

4. Tetris Example

To understand the benefit of segmenting traces based on interface actions, consider a simple interactive piece of software - Tetris. Tetris is a game where the player moves falling games pieces (tetrominoes) of various shapes composed of 4 square blocks to create contiguous horizontal lines. When a contiguous horizontal line of blocks is created, it is deleted and the blocks above move down. The game pieces can be moved left and right by the player and rotated into different orientations. The game ends when the game pieces fill the game area. Although it is a relatively simple application, there is actually quite a bit occurring in terms of the user input, the system's response in the form

Table 2. UI Traces for rotating a block in Tetris

#	UI Traces	Associated Data
1	Launch	
2	DisplayUpdate	Component: GamePanel
3	DisplayUpdate	Component: BoardComponent
4	MouseClick	WidgetID: startButton, (24, 30)
5	DisplayUpdate	Component: BoardComponent
6	ThreadSignal	ThreadID: 12
7	DisplayUpdate	Component: GamePanel
8	DisplayUpdate	Component: BlockFigure
9	ThreadSignal	ThreadID: 12
	...	
22	KeyPress	Key code: SPACE
23	DisplayUpdate	Component: GamePanel
24	DisplayUpdate	Component: BlockFigure
25	ThreadSignal	ThreadID: 12
	...	

of output, and the computation that is performed. During a typical game, the user presses the left and right arrow keys in order to move a falling game piece into the correct position before it reaches the bottom of the screen. They can also press the down arrow key to immediately send the piece to the bottom and start the next piece falling. Pressing another key may rotate a game piece, pause the game, or quit the game altogether. From the user's perspective, pressing the keys causes changes in the game.

From the implementation perspective, there is quite a bit more occurring - input is being processed, computations are being performed, threads are being spawned, and output is being rendered. Depending on the particular implementation it is likely there exists a thread for gathering and handling the user input, and another thread for timing the progress of the game and the falling game pieces, and perhaps another thread for drawing elements of the interface to the screen. If software maintainers working on the Tetris game wants to understand how rotation of a game piece works from the implementation perspective, they might run the application, rotate a piece, and examine the resulting execution trace to get a sense of what components and source code was exercised. A typical trace generated from the rotation use case would consist of approximately 1500 method calls. It is difficult to isolate in the lengthy trace which calls had to do with the actual rotation of the block, and which were related to other activities - such as setting up windows, game state, game timers, etc.

If the execution trace is segmented by the UI actions we have defined in Table 1, we get the list of UI traces seen in Table 2. The Launch and Exit actions are self-explanatory. Following the Launch action, two DisplayUpdate actions occur, one associated with an object of type GamePanel, and the other with an object of type BoardComponent. The MouseClick action is recorded when the Start Button was clicked. At several points a thread signal is recorded followed immediately by DisplayUpdate actions. Later in the trace a KeyPress action is captured. The key pressed was the space bar, which is the key used to rotate the game pieces.

Each UI trace also has associated computation and data. The computation is displayed as the method call tree that occurs in between the beginning and finish of the UI action. The associated data can consist of x, y coordinates, component id's, or thread id's. The third column in Table 1 lists some of the data associated with each UI action.

To determine whether our segmentation approach is useful when performing typical maintenance tasks, we perform the maintenance task of adding new functionality to the Tetris game. Implementing a configurable difficulty setting in the Tetris game will require the following steps: (1) Providing the user a way to manually set the difficulty via the interface; (2) Causing the blocks to fall faster based on the difficulty setting; and (3) Automatically determining when the difficulty setting should be increased (based on total lines cleared).

Loading the Tetris application into the tracing prototype and running it, and playing a short game, generates a UI trace containing a number of paint, thread, and key event actions. UI traces for the launch and initial painting of the user interface and its components are generated, along with thread signal and paint UI traces for figures being generated repeatedly as the game pieces fall. Scrolling through the UI traces we can eventually find the UI trace for when a line is about to be cleared. We can also find the UI trace where the line was cleared by scrolling through the screen shots. When we encounter a screen shot that has the line removed, we know that somewhere in the last UI trace, a method was called that cleared the line. These points in the program are ones that will likely need to be altered in order to implement a changeable difficulty setting. The prototype allows us to quickly identify and navigate to these points with little to no prior knowledge of how the Tetris game is implemented.

A raw execution trace taken from starting Tetris, and playing up until a line is cleared is approximately 14500 method invocations of both native and non-native Java methods. Segmenting this trace results in 1035 UI traces. The Tetris game implementation consists of 11 classes and 85 methods written in 2,418 lines of code. The JDK libraries adds an additional 9 packages, 55 classes, and 1,084 methods. The source code for the Tetris game is available at `http://www.percederberg.net/software/tetris/`.

5. Related Work

Tracing the execution of software systems has been used for some time and many techniques for collecting and storing traces have been developed [6]. The majority of these methods have been used to reduce the cost of collecting and storing the large quantity of data associated with a typical execution of a given program. There have been other approaches to using dynamic system data to understand and visualize software architecture and behaviour. These approaches may describe architecture and the associated behaviour using visual languages [4] or visualize entire traces in the context of the architecture [1, 2, 7]. Our approach is different in that we allow the developer to focus in specific points in a trace based on actions that occur in the interface. Tools that use our approach may use visualizations to present these trace segments.

Walker *et. al.* encode events in an execution trace that allow them to be investigated from different architectural viewpoints [9]. Events such as method execution, object allocations, and thread events are abstracted to types and instances in the architecture. The efficiency of their approach is demonstrated using a visualization tool and a query tool. The way our approach collects execution data is similar to their encoding scheme, however we also segment the collected data based on actual behaviour of the system - the user interface activity.

Eisenbarth *et. al.* used a combination of static and dynamic analysis similar to our approach for an improved means of feature location in source code [3].

6. Future Work

We have also explored integrating the UI trace component of our prototype with a visualization tool that allows the software maintainer to view implementation activity using a visual model. The visual model augmented with UI trace data provides the ability to navigate a large code base while providing information regarding the system architecture as well as details concerning a specific execution of the application [8].

We have used our UI trace prototype on a variety of interactive system maintenance tasks and preliminary indications show that the approach is promising. However, further experimentation is necessary to determine whether the approach scales to larger interactive systems and is applicable for other styles of interactive systems.

Further processing on the UI traces and examination of the patterns that emerge during various uses of software may show various architectural designs are better suited to supporting specific tasks than others.

Integration of this approach into modern development environments (e.g. as an Eclipse plugin) may allow for more extensive evaluation of how navigation in software development can be supported using execution data segmented using interface actions. This may also allow us to determine if the availability of tools using this approach will influence development and architectural design.

7. Conclusion

We have presented an approach for navigating the architecture and source code of interactive software applications based on actions that occur in the user interface. These actions may include user inputs and actions, display updates and drawing operations, and certain processing events that affect the user interface. Executing the target application and tracing the computation that occurs produces large, unwieldy amounts of data. By connecting user interface actions to segments of computation in the trace we produce a UI trace that is easier to navigate and manage. Additionally, this UI trace can be used as a basis for viewing high-level models of system activity. We demonstrate how this approach can assist in performing software maintenance tasks such as implementing new functionality.

References

[1] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58.

[2] P. Dugerdil and S. Alam. Execution Trace Visualization in a 3D space. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 38–43, 2008.

[3] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, March 2003.

[4] J. Grundy and J. Hosking. High-level Static and Dynamic Visualisation of Software Architectures. *Proceedings of SEKE'98, IEEE CS*, pages 18–20, 2000.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, pages 327–353, 2001.

[6] J. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[7] A. Malony, D. Hammerslag, and D. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–28, 1991.

[8] A. Sutherland and K. Schneider. Towards a framework for software navigation techniques. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 101–104. ACM New York, NY, USA, 2008.

[9] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. *ACM SIGPLAN Notices*, 33(10):271–283, 1998.