Design Recovery of Real-Time Graphical Applications using Video

Kim Cuong Pham, Tran Quan Pham, Amir Michail
University of New South Wales
Sydney, Australia, 2052
{kcph007,quanpt,amichail}@cse.unsw.edu.au
UNSW-CSE-TR-0403

Abstract

In a previous paper [4], we introduced an approach to design recovery that takes advantage of the interactive and graphical nature of the majority of today's applications. This earlier work is applicable only to interactive graphical applications written in an event-driven programming style with alternation between user-initiated events and application responses. While productivity applications such as word processors and spreadsheets are of this form, real-time graphical applications such as flight simulators and games are not, since the application proceeds even while the user is idle. In this paper, we propose a design recovery method for real-time graphical applications that uses video to link lower-level code events with their higher-level graphical manifestations. We demonstrate by example how the more easily understood video can shed light on the harder to understand implementation of a real-time graphical application.

1 Introduction

Software maintenance is central to application development. After an application ships, maintenance of the source code is required to correct problems that arise and also to enhance the software. It is estimated that 50%–90% of maintenance effort involves program understanding [16]. One of the primary reasons for this is that often the design of the software system is not well-documented. Moreover, trying to understand design from source code is quite difficult. Consequently, much research has gone into ways for automatically extracting high-level design from code. However, existing research on recovering design from source code has focused on legacy applications, which tend to be batch and text-based. In contrast, today's applications tend to be graphical and event-driven. We propose an approach to design recovery that links lower-level application events with their higher-level graphical manifestations. Other design recovery methods do not make this connection, which can facilitate software maintenance in the following fundamental ways: (1) by identifying a high-level feature visually (e.g., changing text style in a word processor) and then seeing how that feature is implemented by examining the associated low-level event(s); and (2) by explaining the purpose of a low-level event by examining visual displays of the application when that event occurred.

In our initial efforts [4] in linking lower-level code events with their higher-level graphical events, the user uses an application normally while our system automatically picks up user actions using before/after screenshots to describe them visually. In this way, we have a mapping between the low-level code executed in the action with its higher-level visual description in the form of before/after screenshots. Users (perhaps different from those who performed the initial demonstration) can then search and browse the action database for design recovery purposes.

Our preliminary work involves user "actions". In particular, the start of an action is determined by user input while the end is detected when the application goes idle after performing the action. However, this approach is restricted to interactive graphical applications written in an event-driven programming style with alternation between user-initiated events and application responses. In contrast, real-time graphical applications such as flight simulators and games rarely go idle and many important – possibly overlapping – events are not directly initiated by user input. In this paper, we shall present a method for identifying low-level code events that removes the limitations of "actions" to directly support real-time graphical applications.

Our use of before/after screenshots in earlier work for visually describing low-level events, while adequate for productivity applications, presents problems for real-time graphical applications. In particular, before/after screenshots are not as effective when the beginning and end of an event look similar or when there are many overlapping events. In this paper, we shall improve upon before/after screenshots and resolve these problems by using video to

describe the low-level code events.

To demonstrate our approach, we have built a tool, named DRT/Video (for Design Recovery Tool using Video), that works with C/C++ X Window System real-time graphical applications as well as the productivity applications considered in our earlier work. Because our previous tool [4] is named simply DRT, any mention of "DRT" in this paper shall refer to our previous tool for productivity applications while any reference to "DRT/Video" shall refer to the new one for real-time graphical applications.

The rest of the paper is organized as follows. Section 2 describes our technique for finding low-level code events. Section 3 describes how we use video to depict the higher level graphical manifestations of the low-level code events. Section 4 demonstrates how we can take advantage of the linkage between low-level code events and their higher level visual descriptions to perform design recovery of a real-time graphical application. Section 5 discusses related work. Section 6 summarizes the paper and suggests future work.

2 Low-Level Code Events

The low-level code events for DRT/Video differ from the actions in the older DRT tool in several respects: unlike DRT actions, DRT/Video low-level code events may (1) start without being an immediate consequence of user input; (2) end without the application going idle; and (3) overlap with other low-level code events.

For example, in the older DRT tool, consider an action that is detected when the user makes a text selection using the mouse in a word processor. The action is started as a direct consequence of mouse events and ends when the word processor goes idle after executing code to make the text selection. Such actions do not overlap with any other actions.

In contrast, consider an example of a DRT/Video low-level code event in a flight simulator with dynamic weather conditions. An aircraft may fly from a sunny area, pass through a thunderstorm midway, and then arrive at another sunny region. The low-level code event(s) for the thunderstorm: (1) is not an immediate consequence of user input; (2) ends without the application going idle; and (3) overlaps with other low-level code events involved in flight modeling, scenery rendering, etc.

We shall find such low-level code events in two steps: (1) we find "single function events" that denote time intervals during which an execution burst of a *particular function* occurs; and (2) we find "multiple function events" that denote time intervals during which an execution burst of a *set of functions* occurs. For example, a single function event might tell us that thunderstorm::update(...) was called frequently during time interval [25, 300]. A multiple function event might tell us that a set of functions {thunderstorm::update(...), thunderstorm::draw(...), aircraft::simulateTurbulance(...)} was called frequently during the time interval [23, 302]. We shall define more precisely the notions of single function events in Section 2.1 and multiple function events in Section 2.2.

2.1 Single Function Events

Consider a particular function f. Suppose it was called at times $t_1, \ldots, t_n$ throughout an application execution. Roughly speaking, a *single function event* (SFE) occurs for f in time interval $[t_i, t_j]$ when f is called frequently in $[t_i t_j]$. More precisely, we use a constant TIMEOUT to determine when a function is no longer being called frequently, so f will have a SFE in time interval $[t_i, t_j]$ if and only if (a) for $k = i, \ldots, j-1$, the lengths of intervals $[t_k, t_{k+1}]$ are $<$ TIMEOUT; (b) the length of interval $[t_{i-1}, t_i]$ is $\geq$TIMEOUT or $i = 1$; and (c) the length of interval $[t_j, t_{j+1}]$ is $\geq$TIMEOUT or $j = n$.[1]

As we shall compute SFEs for all functions f that have executed in the application demonstration, we shall use a 3-tuple (f, s, e) to indicate that a SFE occurred for function f in time interval $[s, e]$. As an example, suppose a function f is called at times 1.0, 1.8, 2.2, and 3.5 while a function g is called at times 1.5, 2.3, and 3.0. If TIMEOUT=1.0, then we shall obtain SFEs $(f, 1.0, 2.2)$, $(f, 3.5, 3.5)$, and $(g, 1.5, 3.0)$.

In terms of implementation, we use the GNU C/C++ compiler's -finstrument-functions option to have our SFE detection code run on every application function call. We have observed that the SFE detection code has marginal impact on execution speed, so instrumented real-time graphical applications continue to be highly usable for demonstration purposes.

2.2 Multiple Function Events

Roughly speaking, a *multiple function event* (MFE) consists of a maximal set of SFEs with similar start times and also similar end times. More precisely, given a constant TOLERANCE < TIMEOUT, we shall allow the start times to vary by at most TOLERANCE and the end times to also vary by at most TOLERANCE. The requirement TOLERANCE < TIMEOUT ensures that any given function f has at most one SFE in a given MFE. Consequently, we can speak of a set of m functions $\{f_1, \ldots f_m\}$ corresponding to a MFE and it will be understood that there are m corresponding SFEs in such a MFE.

A MFE with functions $\{f_1, \ldots, f_m\}$ will occur in time interval $[t_{\min}, t_{\max}]$ if and only if the following conditions

[1] Observe that it is possible to have an SFE in time interval $[t_i, t_i]$ where f is called only once. Thus, SFEs make sense in productivity applications also where user actions may be very short with some important functions f being called only once per user action.

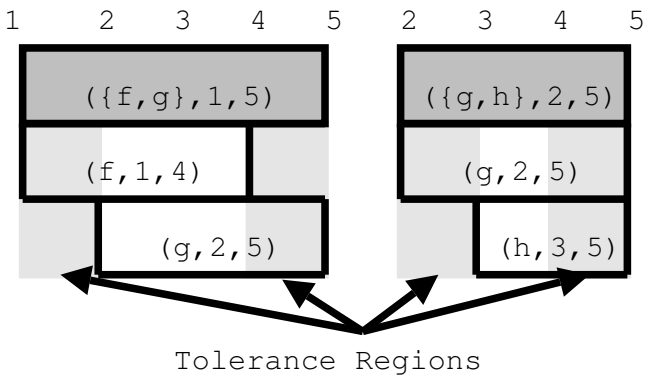


Figure 1. Multiple function events example.

all hold: (1) there is some function $f \in \{f_1, \ldots, f_m\}$ with SFE $(f, t_{\min}, e)$; (2) there is some function $f \in \{f_1, \ldots, f_m\}$ with SFE $(f, s, t_{\max})$; (3) every function $f \in \{f_1, \ldots, f_m\}$ has a SFE (f, s, e) where $s \in [t_{\min}, t_{\min} +$ TOLERANCE] and $e \in [t_{\max} -$ TOLERANCE$, t_{\max}]$; and (4) the MFE must be maximal in the sense that no other MFE satisfying the above conditions contains a superset of this MFE's SFEs

We shall use the 3-tuple $(\{f_1, \ldots, f_m\}, s, e)$ to denote the unique MFE with functions $\{f_1, \ldots, f_m\}$ and time interval $[s, e]$.

Observe that each SFE (f, s, e) will be part of at least one MFE since we will either have a MFE $(\{f\}, s, e)$ or, if this is not maximal, some MFE containing SFE (f, s, e). Moreover, it is possible for the same SFE to be part of multiple MFEs. For example, suppose we have SFEs $(f, 1, 4)$, $(g, 2, 5)$, and $(h, 3, 5)$. If TOLERANCE=1, then we will end up with two MFEs containing $(g, 2, 5)$: $(\{f, g\}, 1, 5)$ and $(\{g, h\}, 2, 5)$. (See Figure 1.) The larger MFE $(\{f, g, h\}, 1, 5)$ is not valid since $(f, 1, 4)$ and $(h, 3, 5)$ have start times that vary by 2, which is greater than the allowed tolerance of 1.

Our algorithm for finding all such MFEs is somewhat involved. A high-level description of the algorithm is given in the Appendix.

3 High-Level Graphical Events

Unlike the older DRT tool that uses before/after screenshots to describe actions, DRT/Video uses video to describe low-level code events. While before/after screenshots are adequate for describing user actions in productivity applications such as word processors and spreadsheets, they are less so for real-time graphical applications.

For example, consider the flight simulator thunderstorm scenario from Section 2. The MFE corresponding to the thunderstorm may start at the very early signs of bad weather and end with the last remnants of the storm. Consequently, it may not be obvious from before/after screenshots that this MFE corresponds to a thunderstorm. This is particularly an issue if the functions in the MFE are more generic weather modeling subroutines that do not contain explicit references to "thunderstorm" in their names. Video is also better than before/after screenshots when there is overlap among events. For example, the aircraft may just happen to start raising its flaps at the beginning of the thunderstorm event, so it would appear from the before/after pictures that perhaps the MFE refers to raising of the flaps. However, it would be obvious from the video that flap raising was completed long before the end of the thunderstorm MFE.

One might wonder whether today's computers are fast enough to do video capture of real-time graphical applications in software. Surprisingly, they are; video capture hardware is not required. On a 1.1 GHz Pentium III computer, we have been able to capture video using the tool xvidcap[2] at 10 frames per second for a window of size 640x480 with only occasional and short-lived slowdowns; real-time graphical applications remained highly usable. However, since frame compression is computationally expensive, we had to perform this video capture by storing uncompressed frames on disk. Of course, once the video has been captured in real-time, compression of the video can be done afterwards to save disk space.

4 Design Recovery

In this section, we shall describe typical usage of DRT/Video.

4.1 DRT/Video Tool User Interface

Figure 2 shows the user interface of the tool. Recall that DRT/Video links low-level code events with their high-level graphical events. As explained in Section 2, the low-level code events are found using MFEs, which in turn are compromised of SFEs. The user interface has two views for low-level code events: (1) an "MFE overview" view that excludes SFEs (shown on left in Figure 2) and (2) a more detailed "MFE contents" view that includes SFEs (shown in the upper right in Figure 2). Also, as explained in Section 3, the high-level graphical events are identified by the user from a video of the application demonstration. Consequently, there is a third view showing the video.

The MFE overview depicts MFEs over time. The *time bar* in the view — the dashed vertical line — indicates the current time in the demonstration. The horizontal position

[2]This tool can be downloaded from http://www.sourceforge.net/projects/xvidcap.

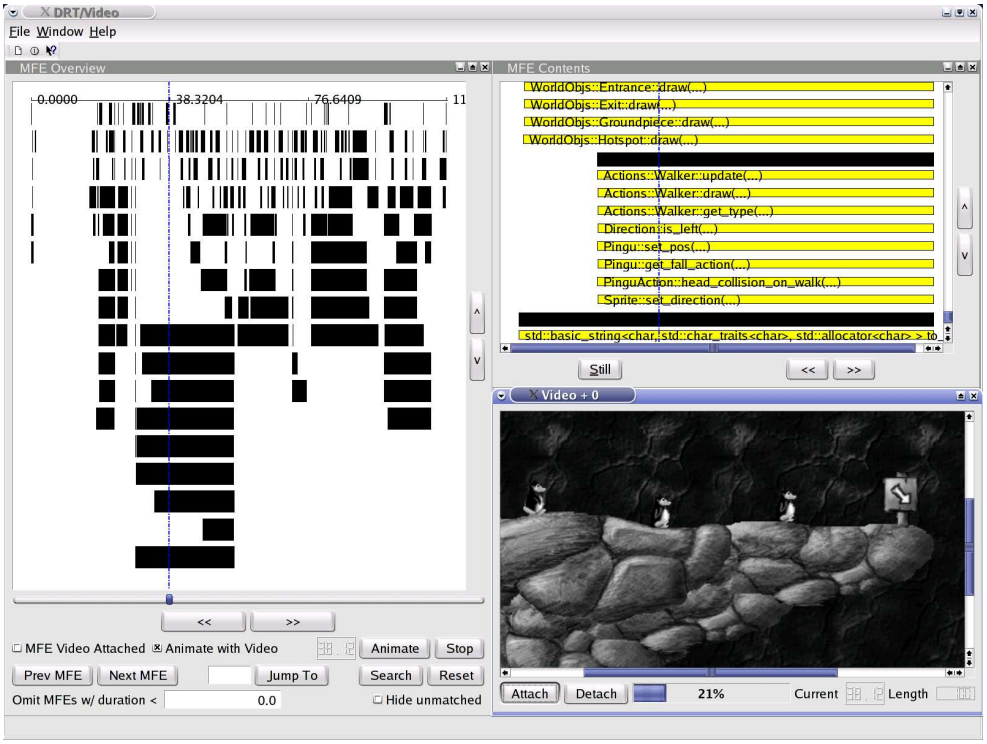


Figure 2. DRT/Video user interface.

and width of a MFE indicates its starting time and duration. However, there is no significance to the vertical position or height of a MFE in this view. We ensure that MFEs overlapping in time are not placed in the same row. Specifically, we lay out MFEs by considering them in non-decreasing order of *end times* and placing each MFE at a vertical position corresponding to the topmost row that is free (i.e., causes no overlap). Observe that this layout algorithm tends to place shorter MFEs in top rows, so if the user is for example looking for a short MFE then it will likely be near the top of the view. The MFE contents view zooms in on only the MFE(s) intersecting the time bar in the MFE overview and shows the SFEs compromising those intersected MFE(s). Note that the width of the SFEs may be slightly smaller than the width of their MFE due to the tolerance parameter. Again, the time bar in this view denotes the current time at this point of the demonstration.

All three views – MFE overview, MFE contents, and video – are synchronized: if the user moves to time t in the demonstration (e.g., using the time slider or one of the "animate", "next MFE", or "prev MFE" buttons), then all three views will correspond to time t. Note also that all three views allow the user to zoom in and out.

4.2 Game Example

To demonstrate typical usage of DRT/Video, we shall consider design recovery of the real-time graphical game Pingus 0.6.0, a Lemmings clone for Linux. In this game, the player needs to guide a group of penguins safely to the exit on each level, while saving them from fatal falls, by giving them commands such as build a bridge, dig a hole, or redirect other penguins in the other direction.

The game makes a good example for several reasons. First, Pingus 0.6.0 is a substantial C++ application consisting of 61,842 lines of code. Second, the game would not work at all with our previous version of DRT. This is because (1) several penguin actions, such as walking and falling, are not directly initiated by user input so the older DRT tool would not be able to detect the start of such events; and (2) code is executed at almost all times since at least one penguin is typically in motion in game play so the older DRT tool would not be able to detect the end of events as there is little if any idle time.

For this demonstration, we played the game Pingus for about a minute and a half under DRT/Video. The video was taken at 10 frames per second. We used a timeout of 0.25 seconds and tolerance of 0.2 seconds, which resulted in 6,169 SFEs and 382 MFEs. As explained in Section 2, the SFEs were identified in real-time while the MFEs were

found after the demo. It took 17 seconds to identify all MFEs using the algorithm described in the Appendix on a 1.1 GHz Pentium III computer.

4.3 Top-Down Design Recovery

In the context of the DRT/Video tool, top-down design recovery means using higher level video events to shed light on their corresponding lower level code events. For example, if we look at the video view in Figure 2, we see penguins walking. Correspondingly, we would expect to find a MFE corresponding to walking. This is indeed the case as is evident from the MFE Contents view, which shows a MFE active at the moment compromised of SFEs Actions::Walker::update(...), Actions::Walker::draw(..), and several other functions involved in walking. DRT/Video currently does not differentiate between instances of the same class, so there is only one MFE active even though there are three penguins walking.

In this level of the game, if the penguins keep walking to the right, they would fall off a tall cliff. To prevent them from dying, the player would need to make them "floaters" – meaning that they would be given a propeller to slow them down during the fall. This is done before the fall while they are still walking towards the cliff edge.

If one is not familiar with the code at all, then it is not obvious which functions would be executed in this chain of events. Even if one has briefly looked at the code and noticed that there are classes for walkers, fallers, and floaters, questions still remain. When a penguin is made into a floater while it is still walking towards the cliff edge, is it considered a walker, a floater, or both? When the penguin starts falling, is it considered a faller, a floater, or both?

Figure 3 depicts this chain of events. In particular, Figure 3, (a) shows the penguin – already converted to a floater – walking towards the drop. The MFE contents view tells us that it is considered a walker at this point since Actions::Walker::update(...) and Actions::Walker::draw(...) are being repeatedly called and no other penguin is walking at that point in the video. (Incidentally, Actions::Splashed::update(...) and Actions::Splashed::draw(...) refer to a penguin just in front that fell to its death without a propeller.) Figure 3, (b) shows the initial drop of the floater penguin. The MFE contents view tells us that it is actually considered a faller at this point with functions such as Actions::Faller::update(...), Actions::Faller::draw(...), and a host of others compromising the fall MFE. Figure 3, (c) shows that soon afterwards the propeller takes effect and the penguin turns into a floater, with functions Actions::Floater::update(...), and Actions::Floater::draw(...) being executed.

Observe that without the video guiding our analysis of the code, it would be more difficult to understand the precise scenarios in which penguins are transformed between various types. The video is higher level and more easily sets the context for the lower-level code events.

4.4 Bottom-Up Design Recovery

In the context of the DRT/Video tool, bottom-up design recovery means shedding light on lower level code events by observing their corresponding higher level video events.

However, while it is easy to simply view an entire video from start to end to identify interesting higher-level visual events for top-down design recovery, it would be much more difficult to consider all MFEs manually to identify interesting lower-level code events. Consequently, it is important to provide the user some way of finding MFEs of interest. The DRT/Video tool supports keyword search for this purpose. Figure 4 shows the result of the keyword query "faller". In the MFE overview, shaded MFEs indicating matching MFEs with at least one SFE containing the query word(s) in the function name. The darker the shading, the better the match (in a cosine similarity sense as discussed below) with white MFEs indicating no match at all.

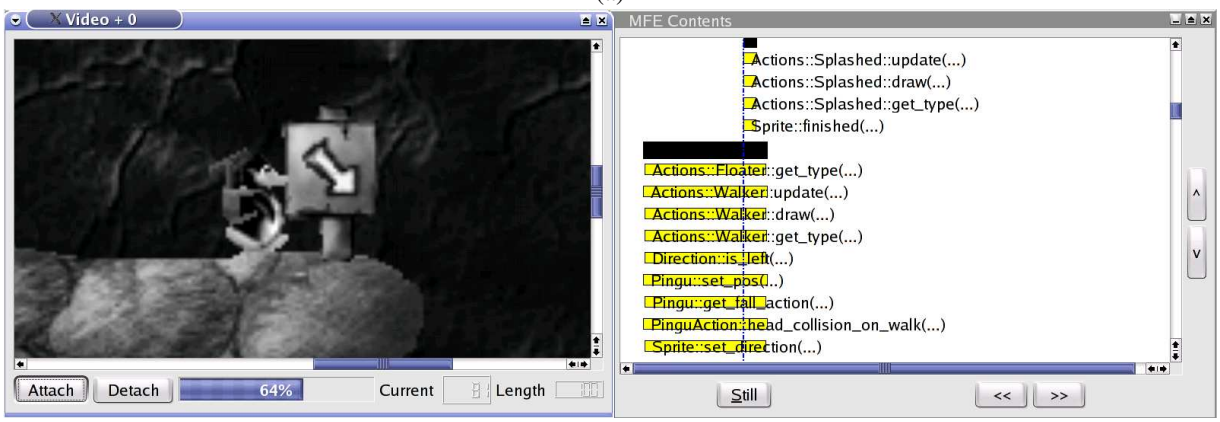
In this example, examining MFEs with "faller" in them would allow us to make a similar observation as that done in Section 4.3 with respect to falling floater penguins. Specifically, we would notice that the matching MFEs seem to be divided into two groups: those with short durations and those with long durations. (See Figure 4.) And so, we can use the video to shed light on those two groups in a bottom-up fashion. In doing so, we would observe from the video that the longer matching MFEs correspond to penguins falling to their death while the shorter ones correspond to an initial quick short fall followed by activation of the propeller to slow the penguin down (in which case it is then considered a floater).

One problem with the bottom up approach using keyword queries is that it requires some initial familiarity with the source since the user needs to know the precise words to use in the keyword query – although stemming is done to alleviate this problem somewhat. For example, should we search for "faller", "jumper", or "diver"?

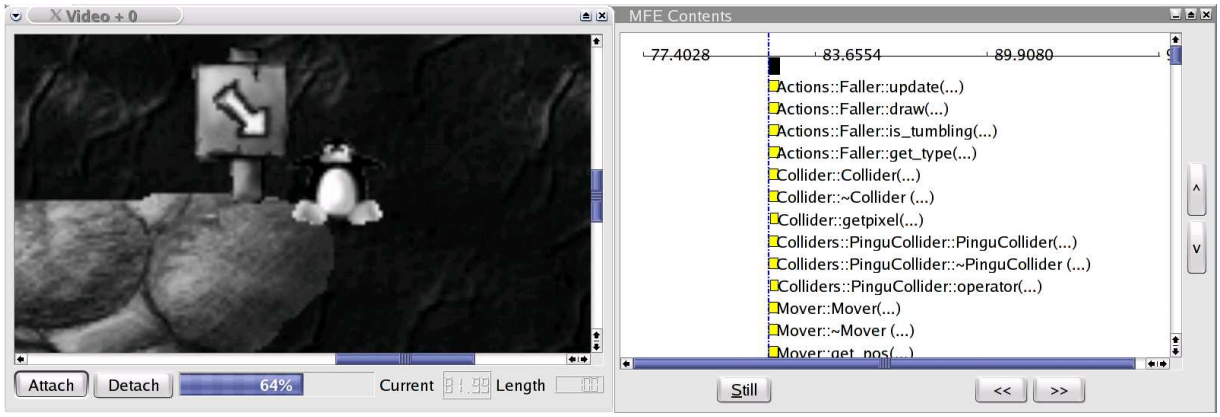
DRT/Video also allows the user to click on a MFE to find similar MFEs. In this way, the user can find related code, which can then be explained visually through corresponding video segments.

For both keyword queries and finding similar MFEs, there is a *query function set* Q: (1) for a keyword query, Q consists of all functions containing all the query keywords in their names; and (2) for a similar MFEs query, Q consists of all the functions in the MFE clicked by the user. DRT ranks MFEs related to this query function set using the standard cosine similarity measure used in information retrieval [21, p. 185]. In particular, if we let M denote the

(a)



(b)



(c)

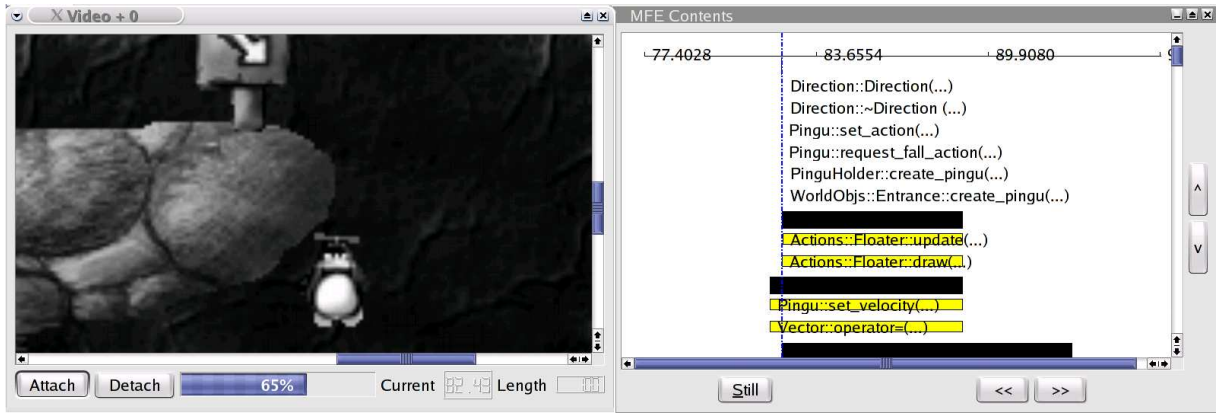


Figure 3. A penguin with a propeller walking off the cliff edge.

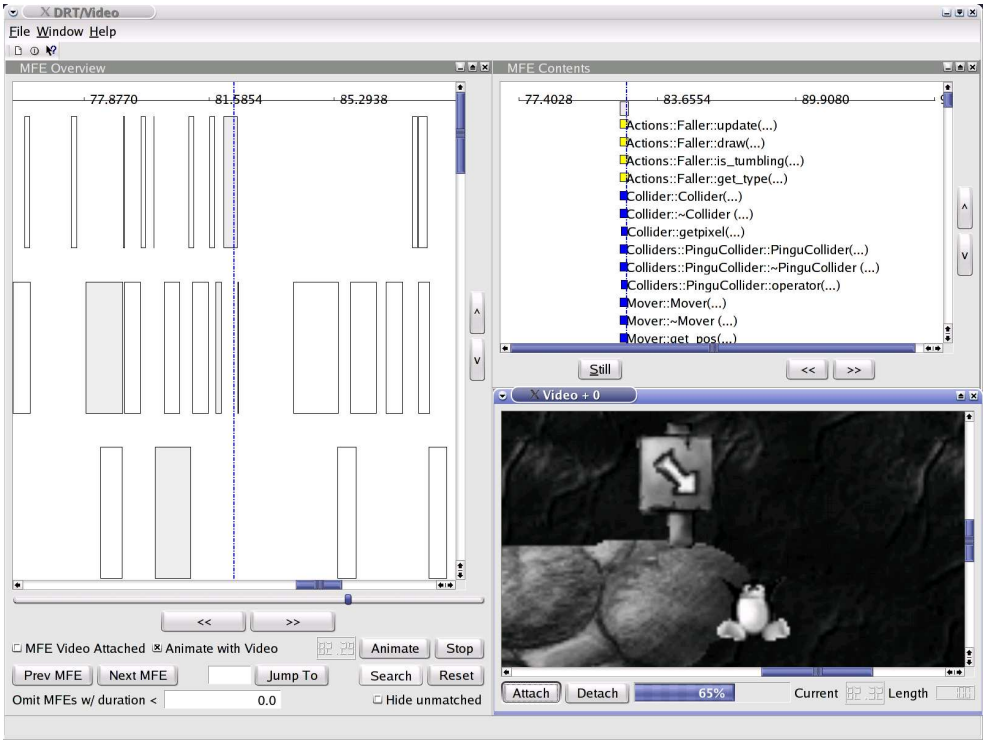


Figure 4. The results of the keyword query "faller".

set of functions in some MFE, then the cosine similarity between Q and the M is $\frac{\sum_{f \in Q \cap M} w(f)^2}{\sqrt{\sum_{f \in Q} w(f)^2}\sqrt{\sum_{f \in M} w(f)^2}}$, where we set weight $w(f)$ to one.

5 Related Work

As explained in Section 1, our earlier work on DRT [4] was applicable only to interactive graphical applications written in an event-driven programming style with alternation between user-initiated events and application responses. DRT/Video removes this restriction to fully support real-time graphical applications. Other related work follows below.

5.1 Software Visualization

Since DRT/Video uses video to visually describe low-level code events, one can view the tool as a software visualization system. Other work on software visualization typically involves the development of new visualization techniques —- independent of the application display — whether they be visualizations of the source code itself [1] or perhaps visualizations of more abstract entities such as algorithms and data structures [3, 10].

5.2 Macro Recorders and Programming by Demonstration

Macro recorders allow users to record their actions and replay them later [6]. For example, a spreadsheet user can automate the selection of certain cells and subsequent copying to another location. DRT/Video also allows users to record a demonstration of the application but for the purposes of design recovery; actions are not replayed later to a running application.

Macro recorders record the user's actions verbatim. In contrast, a programming by demonstration system creates generalized programs — possibly with conditionals and loops — from the recorded actions [6]. For example, such a tool may generalize from the actions of a spreadsheet user that it should select all of the rows before row "Total". As another example, the SMARTedit system [12, 13] can generalize editing tasks such as deleting HTML comments from examples. Such generalizations can become more accurate when multiple demonstrations are given by the user for a particular task so that incorrect inferences are filtered out. The DRT/Video tool does not currently do any such generalization, but one can imagine adding clustering to discover more abstract code events (e.g., the notion of a penguin action in the game Pingus whatever that action may be).

The work on editable graphical histories [11] introduces the notion of before/after pictures to identify past actions in a macro recorder or programming by demonstration system. As an example of this approach, the authors developed a tool that captures the user's actions in a drawing program using before/after pictures so the user could easily go back to some previous point in the session, edit the drawing at that point, and then replay subsequent drawing operations performed in the past. DRT/Video is similar in the sense that it uses video to visualize the application state at a very high level of abstraction.

5.3 Software Clustering

Much of the design recovery literature involves software clustering. Software clustering attempts to automatically decompose software systems into meaningful subsystems to facilitate understanding of those systems (e.g., for maintenance tasks) or to reuse subsystems in other applications [2, 5, 15, 18].

Our approach does not result in a hierarchical decomposition of subsystems. Rather, we obtain a collection of execution traces through the application — namely, the MFEs whose SFEs may crosscut multiple subsystems. However, it is possible to visually identify hierarchical relationships in the MFE overview view in the sense that certain shorter MFEs may be seen to always lie within the time intervals of certain longer MFEs.

Software clustering techniques can be divided into two categories: *knowledge-based* and *structure-based* approaches [18]. Knowledge-based approaches attempt to understand what different pieces of source code do by using reverse engineering techniques and pre-existing domain knowledge [2, 15]. For example, such a technique may recognize clusters at a conceptual level, such as a cluster of functions responsible for "reserving an airline seat".

Structure-based approaches look at simpler syntactic interactions between objects [5, 18]. For example, the objects may be functions while the syntactic interactions may be function calls. Structure-based algorithms typically identify clusters based on maximization of cohesion and/or minimization of coupling. Most software clustering researchers have concentrated on structure-based techniques as they do not require building of a knowledge base or problem domain semantics [18].

Our approach does not require any knowledge of the real-time graphical application in particular (e.g., flight simulator); rather, it requires knowledge of real-time graphical applications in general (e.g., many interesting events involve repeated invocation of a function over a time interval and have obvious graphical manifestations over that same time interval).

5.4 Dynamic Analysis

There has been related research done on selective tracing of atomic units of activity. For example, the drive-by analysis approach employed in the Jinsight tool [14] allows users to define their own notions of execution "burst" by specifying triggers, filters, requested threads, and burst ending criteria. Jinsight returns dynamic information from the burst only, thus allowing users to prune potentially very large amounts of trace data to only those items of interest. We also have a notion of an execution burst in the DRT tool – namely in the form of SFEs and MFEs – but we predefine it as explained in Section 2 to automatically capture bursts that correspond closely to interesting events that occur in a real-time graphical application. While Jinsight does share the generic idea of an execution burst with DRT/Video, it does not automatically find bursts (without user input) nor does it describe them visually from the application display.

However, there is an interaction-driven tool, LeNDI, for reverse engineering legacy interfaces [7, 17] that identifies the text screens encountered while an application is used and ways (e..g, keyboard input) for getting from one text screen to another. Such information can be helpful in migrating text-based legacy interfaces to newer platforms (e.g., GUIs or the web) or perhaps wrapping them for interaction with other systems. While it may seem that LeNDI is similar to DRT/Video in the way it uses the application display, LeNDI actually makes no connections to the corresponding burst code. The authors acknowledge that LeNDI does not shed much light on the application code behind those text-based interfaces [17]: "Clearly, if the end goal is to extend or modify the system functionality by modifying its current code, then such interaction-based understanding is insufficient because it provides only a model of the user tasks that the system supports and it completely ignores low-level design decisions such as data structures and algorithms."

The "software reconnaissance" technique involves running test cases with and without the desired feature and looking for code executed in the test case(s) with the feature that is not executed in the test case(s) without it [19, 20]. Unlike "software reconnaissance", DRT/Video does not require the user to manually set up various test cases with and without certain features. Rather, the user simply demonstrates the application normally exercising a variety of features. Video and MFEs provide the connection between features and implementation.

Anomaly detection, much like MFEs, identifies interesting low-level events throughout the application execution but these anomaly events represent abnormal/buggy behavior that should be corrected [9]. Consequently, unlike our approach, anomaly detection involves a training phase to identify expected behavior followed by a checking phase to

see if any abnormal behavior is present.

6 Conclusions and Future Work

In this paper, we have explored a method for design recovery of real-time graphical applications using video. The method involves linking low-level code events with their higher level graphical manifestations. We have described how to identify low-level code events in a two step procedure: first, we identify single function events that correspond to time intervals during which a particular function is called frequently; and second, we identify multiple function events that are compromised of SFEs all of which have similar start and end times. The higher level graphical events can be identified by simply watching a video of the application demonstration. We have demonstrated how the linkage between the low-level code events and the high-level graphical events can be used for both top-down and bottom-up design recovery.

For future work, we would like to support heavily data-driven real-time graphical applications. In such applications (e.g., a physical simulation), the control flow is very similar throughout the execution, so we would not pick up many interesting MFEs. However, interesting low-level events can be found if we look at the program state over time, perhaps using invariant mining [8]. For example, in a flight simulator, when an airplane starts to descend, there would be a corresponding low-level event indicating that certain variables are now negative say. However, we would need to look for invariants that hold only during a limited period of time – much like work done on anomaly detection using invariant mining [9].

We would also like to create "video hyperlinks" to more directly link visual objects in the video with their corresponding low-level code events. This would allow one to attach to a visual object to trace its execution much as one would attach to a thread or process in a debugger. For example, in a basketball game, one could click on the ball in the video to see the associated code for the physical simulation of the ball – even in the presence of other moving objects such as the players, referees, and spectators. The video hyperlink can also be used in the other direction. The user can click on a low-level code event to highlight the visual object(s) in the video to which the event corresponds. So if one clicks on some low-level physical simulation code event(s) and sees that the ball is highlighted in the video, then it would be immediately clear that this code is responsible for physical simulation of the ball. To create video hyperlinks, we could look for objects in the video segment that undergo some transformation (e.g., changes in location, color, texture, shape, etc.) over a time interval that coincides with that of a low-level code event.

References

[1] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, 1994.

[3] M. Brown. Zeus: A system for algorithm animation and multi-view editing. *Computer Graphics*, 18(3):177–186, 1992.

[4] K. Chan, Z. Liang, and A. Michail. Design recovery of interactive graphical applications. In *Proceedings of the 25th International Conference on Software Engineering*, pages 114–124, 2003.

[5] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28:117–127, 1995.

[6] A. Cypher, editor. *Watch What I do: Programming by Demonstration*. MIT Press, 1993.

[7] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, and B. Matichuk. Modeling the system-user dialog using interaction traces. In *8th Working Conference on Reverse Engineering*, pages 208–217, 2001.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, 2001.

[9] S. Hangal and M. S. Lam. Tracking down bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[10] R. R. Henry, K. M. Whaley, and B. Forstall. The University of Washington illustrating compiler. In *PLDI*, pages 223–233, 1990.

[11] D. Kurlander and S. Feiner. Editable graphical histories. In *IEEE Workshop on Visual Languages*, pages 127–134, 1988.

[12] T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *17th International Conference on Machine Learning*, pages 527–534, 2000.

[13] T. Lau and D. S. Weld. Programming by demonstration: An inductive learning formulation. In *International Conference on Intelligent User Interfaces*, pages 145–152, 1999.

[14] W. D. Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivan. Drive-by analysis of running programs. In *Proceedings for of ICSE 2001 Workshop on Software Visualization*, pages 17–22, 2001.

[15] R. C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[16] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.

[17] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk. Reverse engineering legacy interfaces: An interaction-driven approach. In *6th Working Conference on Reverse Engineering*, pages 292–301, 1999.

[18] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[19] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *International Conference on Software Maintenance*, pages 312–318, 1996.

[20] N. Wilde and M. S. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[21] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

Appendix

At a high-level of abstraction, our algorithm for finding MFEs is as follows:

1. SFE input set I :={all SFEs found in execution}; SFE active set $A := \emptyset$; MFE result set $R := \emptyset$
2. while SFE input set $I \neq \emptyset$ or SFE active set $A \neq \emptyset$:
   (a) if $I = \emptyset$ or SFE (f_I, s_I, e_I) with minimum end time in I has $e_I >$ TOLERANCE + e_A where (f_A, s_A, t_A) is SFE with minimum end time in A:
      i. add MFE(s) containing (f_A, s_A, e_A) (and possibly other SFEs) in A to R but omitting those MFE(s) that are subsets of MFEs already in R
      i. remove (f_A, s_A, e_A) from A
   (b) else: *// SFE (f_I, s_I, e_I) with minimum end time in I has $e_I \leq$ TOLERANCE + minimum end time of SFEs in A (if any)*
      i. add SFE (f_I, s_I, e_I) to A
      i. remove SFE (f_I, s_I, e_I) from I

The algorithm works as follows. We maintain three sets: (1) the *SFE input set* I; (2) the *SFE active set* A; and (3) the *MFE result set* R. Any given SFE (f, s, e) will be in one and only one of these sets, starting out in the SFE input set I, then moving to the SFE active set A (in step 2, (b)), and finishing off in the MFE result set R (possibly as part of multiple MFEs in R) (in step 2, (a)).

To see why the algorithm works, observe that we would only construct MFEs containing SFEs with start times and end times within the tolerance limits in step 2, (a), i, so the output will only contain valid MFEs provided that they are maximal. The condition for whether to move an SFE from the SFE input to the SFE active set or from the SFE active set to the MFE result set ensures that all SFEs in the active set A have end times within TOLERANCE of each other, and so we need only check whether the start times are also within TOLERANCE when generating MFEs. The reason why we may need to generate more than one MFE containing (f_A, s_A, e_A) is because SFEs may have start times within TOLERANCE of SFE (f_A, s_A, e_A)'s start time but up to 2×TOLERANCE from each other's start times.

To see that the MFEs generated in step 2, (a), i are maximal, consider any maximal MFE $M = (\{f_1, \ldots, f_m\}, s, e)$ that should be in the MFE result set R. Consider the function f whose SFE (f_M, s_M, e_M) in M is first processed in step 2, (a). Since SFEs are considered in order of non-decreasing end times, it follows that f_M's SFE end time is $\leq$ the end times of other SFEs in M. Moreover, when SFE (f_M, s_M, e_M) is processed in step 2, (a) all SFEs in M are in the set A since their end times vary by at most TOLERANCE and we do not process SFE (f_M, s_M, e_M) in step 2, (a), i until we are sure that there is no SFE remaining in the input set whose end time is $\leq e_M +$ TOLERANCE. Since (f_M, s_M, e_M) is the first SFE in M to be processed in step 2, (a) it follows that no MFE containing a subset of the SFEs of M has already been put in the MFE result set R. Consequently, the maximality test passes and we add exactly MFE M to the result set R. When the other SFEs in M are processed later, the maximality test will fail and we will not add any subset MFEs of M.

The actual implementation used is a bit more complex since some optimizations are made. For example, we make the maximality check more efficient by observing that for any MFE in R, there comes a point in time at which that MFE could not possibly be a superset of any future MFEs considered for addition to R.