smid: A Black-Box Program Driver

Kareem Khazem[1(✉)] and Michael Tautschnig[2]

[1] University College London, London, UK
karkhaz@karkhaz.com
[2] Queen Mary University of London, London, UK

Abstract. We aim to perform dynamic analysis at large scale and across a wide range of programs. A key problem to overcome is driving interactive programs effectively and efficiently. To address this problem, we developed SMID—an open-source tool that autonomously interacts with computer programs, based on a specification of which user interactions (key presses, mouse events) are valid. Users can define the space of valid user interactions as a state machine in the SMID language. SMID can then generate 'sensible' program runs (sequences of user interactions), which it sends to the target program. Runs can be saved and played back, facilitating the reproduction of bugs. We have used SMID to reproduce and help explain a bug in the CMUS music player. It is possible to use SMID to drive a wide variety of desktop programs, including those with a graphical, console, or even non-interactive user interface.

1 Introduction

Multiple efforts at automated *static* analysis of large software repositories exist, and have had a measurable impact on software quality. Notably, the experiment described in [9] has resulted in hundreds of bug reports, many of which have since been closed as fixed; and the Debile[1] effort has provided developers with the results of running several static analysers through a uniform interface.

We wished to have a tool with several features that would enable similar experiments to be run with *dynamic* program analyses. These features are:

Automation. We require the ability to automatically drive the user interfaces of computer programs.

Wide Applicability. We require that our tool can be used to drive a wide variety of software, regardless of the high-level user interface toolkit used.

Reproducibility. If a certain sequence of user inputs gives rise to an interesting behaviour (e.g. a crash), we require the ability to 'play back' that sequence in order to reproduce the behaviour.

Realism. We do not wish to spam target programs with random inputs, but would like to be able to exercise target programs with only those sequences of user inputs that a real user might issue.

Conciseness. On the other hand, we do not wish to define every possible use case as a set of user interaction 'scripts', but would rather have a concise definition of the desired user interactions with the target program.

[1] http://debile.debian.net.

D. Bošnački and A. Wijs (Eds.): SPIN 2016, LNCS 9641, pp. 182–188, 2016.
DOI: 10.1007/978-3-319-32582-8_12

1.1 Overview

We have implemented SMID (the *state machine interface driver*)—an open-source[2] tool that autonomously interacts with desktop computer programs, driving them with user input in the same way that a real user might. In this paper, we outline the aims and motivation of the tool in Sect. 1; describe SMID's input language in Sect. 2; and describe the tool's usage and how we used it to reproduce a bug in Sect. 3. A video demo of the tool can be viewed on YouTube[3].

With SMID, the space of valid interactions with a program is understood as a state machine—that is, a directed graph whose nodes are *states* and whose transitions are *sequences of user interactions*. A state is a point during the program's execution where a certain set of transitions are possible; these are often referred to as 'modes' in the user interaction literature [12].

Example: In a graphical web browser such as Firefox or Chromium, the main browser window, as well as each of the auxiliary windows (download window, print dialog, etc.) can be understood as states. Transitions (key presses and mouse events) may cause the state to remain the same (e.g. when pressing `Ctrl+R` to reload a web page), or change the state (e.g. clicking on the 'Options' menu item changes the state to the Options window). Different views in the same window may also be understood as separate states.

To use SMID, users describe such a state machine by writing a file in the SMID language. The file is parsed by the SMID tool, which outputs a single 'run' (a finite path through the state machine, from the initial to a final state). SMID then autonomously 'drives' the program by 'playing back' this run—that is, by sending the sequence of user interactions to the target program. By playing back a large number of runs, the target program can be rigorously tested.

1.2 Comparison with Existing Tools

Tools that we have explored, both in academia and in the wider software development community, fell short of one or more of the requirements listed above.

In both academia and industry, the majority of black-box driving tools are only able to interact with programs using one particular user interface toolkit (UIT). The tools Selenium [15], Mozmill [14], Watir [11] and Canoo [8] can only be used to autonomously interact with web applications. EXSYST [7] is a Java GUI automator, although its search-based technique is not limited to Java programs. A tool that does not have the single-UIT limitation is Jubula [16], which is able to drive applications using one of several web and desktop UITs.

Jubula and the web application drivers mentioned above work by executing a 'script' of user interactions, which corresponds to a single use case. To test the target program on all use cases, one must write a large number of scripts; changes in the underlying program then require updates to all affected scripts. We find this method to be fragile, and mention how SMID negates the need for redundant

[2] https://github.com/karkhaz/smid.
[3] https://www.youtube.com/watch?v=x45jjr5dIiY&feature=youtu.be.

specification in Sect. 3. There has been some work in academia on *automatically* determining the behaviour of the target program. Amalfitano et al. [1,2] describe techniques for determining the behaviour of Android applications and web applications, respectively. A formal approach to designing, specifying, and subsequently generating tests for graphical user interfaces is described in [3]. There has also been work to automatically understand user interface behaviour using techniques from artificial intelligence [10] or machine learning [5]. Nevertheless, these techniques have each been implemented for only a single UIT. One possibility for future work is to use the Linux Desktop Testing Project [4] (LDTP) to provide the highly semantic information that is needed for these 'behaviour-learning' tools, for a wide range of UITs and platforms.

In contrast, SMID allows the specification of the set of user interactions that are sensible for the target program using the language described in Sect. 2, without burdening the user with writing interaction 'scripts' and repeatedly specifying common interaction sequences. The fact that SMID sends events directly to the X Window System—which underlies all graphical user interface toolkits on desktop Linux and BSD operating systems—means that SMID is able to drive all interactive desktop applications, as well as console-based applications (through interaction with a terminal emulator), that run on an X-based operating system. The SMID language itself is UIT-agnostic, and we hope to use LDTP in the future in order to provide SMID users with interactions that are more semantic than raw interactions with X—for example, by specifying graphical widgets by name rather than by coordinate.

1.3 Scope and Limitations

SMID is aimed at running experiments on the large body of software supplied with a Linux distribution. Accordingly, while the language described in Sect. 2 is UIT-agnostic, our implementation of SMID targets desktop applications; we did not attempt to implement driving of touch- or voice-based user interfaces. The back-end to our tool is XDOTOOL [13], an API for sending events to an X server, but we expect that SMID can be modified to use a different back-end (like LDTP) in order to make it usable on other systems. While SMID is used to drive target programs toward error states, SMID does not implement any error detection itself; we leave application-specific error-detection to the user.

2 The SMID Language

The SMID language is used to describe a user interface as a state machine. Files written in the SMID language consist mostly of transition statements; in this section, we describe the various forms that transitions can take, as well as other features of the SMID language. The most up-to-date guide to the language is the reference manual[4] found on the SMID web site.

[4] http://karkhaz.com/smid/refman.html.

Figure 1 is a minimal SMID file. It shows the five kinds of statements in the SMID language: *initial* and *final* state declarations, a *transition* (containing *actions*), and *region* declarations. The states in the state machine are all states declared as the start or end state of a transition (i.e., SMID checks that there is a path from the initial state to a final state through every state)—in Fig. 1, those are `nice_state` and `boring_state`.

```
initial nice_state
final boring_state

nice_state --
  keys  [ Control+b Return ]
  text  "A random string"
  move  zork
  click (1 right)
--> boring_state

region zork = ( 30 110 50 145 )
```

Fig. 1. A minimal SMID file

The indented lines from `nice_state` to `boring_state` in Fig. 1 describe a transition. The syntax for transitions is `start_states -- list of actions --> end_state`. At any point during SMID's execution, SMID has a 'current state'. At each step of the program run, SMID chooses a random transition enabled at the current state; it then performs the actions of that transition, and then changes the current state to be the end state of the transition. The possible actions include `keys`, `text` and `line` for sending keypresses to the target program; `move`, `move-rel`, `click` and `scroll` for mouse events; as well as actions for switching windows, executing shell code, and changing the probability of following transition (SMID performs a random walk over the state machine by default). Complex user interfaces beget large numbers of similar transitions, so the SMID language includes syntactic sugar to make it possible to represent several transitions in a single statement—shown in Figs. 2 and 3. This allows us to specify large state machines using fewer statements, as noted in Sect. 3.

```
all -- keys [ q ] --> quit

foo -- keys [ q ] --> quit
bar -- keys [ q ] --> quit
baz -- keys [ q ] --> quit
```

Fig. 2. The first line is equivalent to the next three taken together, if `foo`, `bar`, `baz` and `quit` are the only states in the state machine and `quit` is a final state.

```
foo bar -- keys [ r ] --> stay

foo -- keys [ r ] --> foo
bar -- keys [ r ] --> bar
```

Fig. 3. The first line is equivalent to the next two taken together.

3 SMID Usage and Case Study

Given a SMID file with the syntax described in Sect. 2, one can use the SMID tool for several functions:

Visualising the State Machine. SMID can generate a diagram representing the state machine described by a SMID file. Figure 5 shows a state machine diagram generated from the SMID file in Fig. 4.

Generating a Run. SMID can output a list of user actions, called a 'run,' by accumulating the actions along a finite-length walk of the state machine.

Playing Back a Run. SMID can read a run and sends the specified interactions to the target program. Thus, given a run, SMID can autonomously interact with the target program in the same way a real user might.

```
initial new_window
final quit

all -- keys [ Control+Q ] --> quit
all -- keys [ Control+N ] --> new_window

all-except print_dialog -- keys [ Control+P ] --> print_dialog

new_window load_window --
  move url_bar
  click (1 left)
  line "URLs.txt"
  keys [ Return ]
--> load_window

load_window -- keys [ Control+S ] --> save_dialog
save_dialog -- keys [   Return  ] --> new_window
save_dialog -- keys [   Escape  ] --> load_window

region url_bar = (30 5 200 15)
```

Fig. 4. A SMID file containing several states.

In this section, we describe using SMID to pinpoint crashes in the target program.

We wrote a SMID file (available on GitHub[5]) for the CMUS[6] console music player. This file had 25 transitions, which SMID expanded into a 103-transition state machine. Hence, we find that the SMID language allows us to specify a wide range of behaviour in a concise format and with little redundancy.

We used this SMID file to reproduce a reported[7] segmentation fault. The bug report suggested that the bug was triggered when playing MP4-encoded media files. We set up a large library of audio and video containers, and a SMID specification designed to hone in on the reported bug. SMID caused CMUS to browse to one of the media files, seek to a random point in the file, and play the

[5] https://github.com/karkhaz/smid/blob/master/state-machines/cmus.sm.
[6] https://cmus.github.io.
[7] https://github.com/cmus/cmus/issues/204.

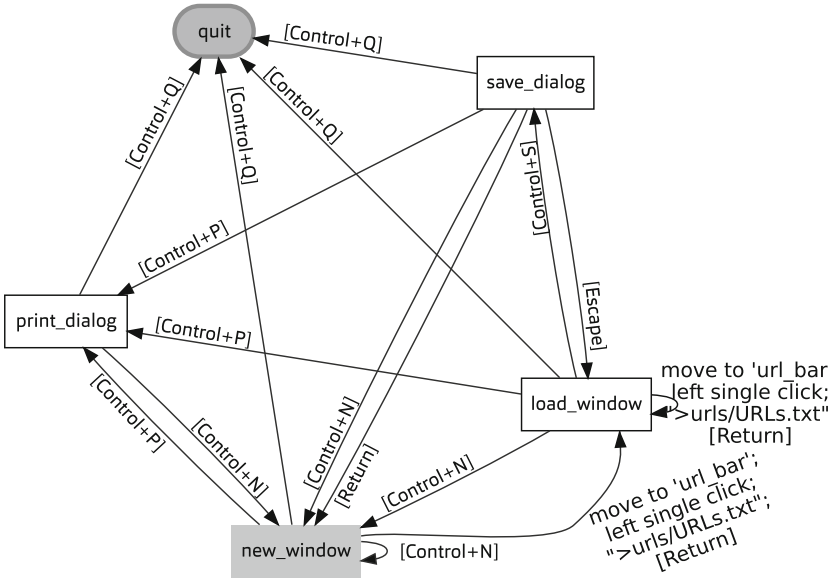

Fig. 5. State machine diagram corresponding to the SMID file in Fig. 4.

data for several seconds. Using this setup, SMID triggered the segfault in many of the several hundred runs that we ran.

By logging CMUS using the SYSTEMTAP [6] instrumentation framework while running it with SMID, we were able to discover several scenarios under which this bug was triggered, but which were not described in the original bug report. This demonstrates the utility of our approach—namely, sending a diverse range of inputs to the target program, from the space of 'sensible' user interactions.

4 Conclusions

The SMID language, described in Sect. 2, is UIT and platform agnostic. The current implementation of the SMID tool uses this fact to drive programs by sending user interactions directly to the underlying window system, rather than to a specific UIT. This means that we are able to drive the large variety of applications that can render a window under the X Window System.

Existing approaches either try to learn the state machine—tying them down to particular UITs, or drive the interface using a script—an approach which is not scaleable. The case study in Sect. 3 shows the value of our approach: by specifying all reasonable behaviours of the target program, we were able to quickly hone in on a bug without spamming the target program with unrealistic inputs.

Acknowledgements. We thank Carsten Fush and Tyler Sorensen, as well as several anonymous reviewers, for constructive feedback.

References

1. Amalfitano, D., Fasolino, A.R., Tramontana, P.: Reverse engineering finite state machines from rich internet applications. In: WCRE (2008)
2. Amalfitano, D., Fasolino, A.R., Tramontana, P., Carmine, S.D., Memon, A.M.: Using GUI ripping for automated testing of Android applications. In: ASE (2012)
3. Berstel, J., Crespi-Reghizzi, S., Roussel, G., Pietro, P.S.: A scalable formal method for design and automatic checking of user interfaces. In: ICSE (2001)
4. Chen, E., LDTP contributors: Linux desktop testing project. http://ldtp.freedesktop.org/
5. Dan, H., Harman, M., Krinke, J., Li, L., Marginean, A., Wu, F.: Pidgin crasher: searching for minimised crashing GUI event sequences. In: Goues, C., Yoo, S. (eds.) SSBSE 2014. LNCS, vol. 8636, pp. 253–258. Springer, Heidelberg (2014)
6. Eigler, F.C., Stone, D., Stone, J., Wielaard, M., SystemTap contributors: SystemTap. https://sourceware.org/systemtap/
7. Gross, F., Fraser, G., Zeller, A.: EXSYST: search-based GUI testing. In: ICSE (2012)
8. Huber, M., Schlichting, M., Canoo contributors: Canoo WebTest. http://webtest.canoo.com/
9. Kroening, D., Tautschnig, M.: Automating software analysis at large scale. In: Hliněný, P., Dvořák, Z., Jaroš, J., Kofroň, J., Kořenek, J., Matula, P., Pala, K. (eds.) MEMICS 2014. LNCS, vol. 8934, pp. 30–39. Springer, Heidelberg (2014)
10. Memon, A.M., Pollack, M.E., Soffa, M.L.: Using a goal-driven approach to generate test cases for GUIs. In: ICSE (1999)
11. Pertman, J., McHowan, H., Rodionov, A.: Watir. http://watir.com/
12. Raskin, J.: The Humane Interface: New Directions for Designing Interactive Systems. ACM Press/Addison-Wesley Publishing Co., New York, NY (2000)
13. Sissel, J.: xdotool–fake keyboard/mouse input, window management, and more. http://www.semicomplete.com/projects/xdotool/
14. Skupin, H., Hammel, J., Rogers, M., Mozmill contributors: Mozmill. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Mozmill
15. Stewart, S., Selenium contributors: Selinium WebDriver. http://www.seleniumhq.org/projects/webdriver/
16. Tiede, M., Struckmann, S., Mueller, M., Jubula contributors: The Jubula functional testing tool. http://www.eclipse.org/jubula/