GROPG: A Graphical On-Phone Debugger

Tuan Anh Nguyen, Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, TX 76019, USA
tanguyen@mavs.uta.edu, csallner@uta.edu

Nikolai Tillmann
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
nikolait@microsoft.com

***Abstract*—Debugging mobile phone applications is hard, as current debugging techniques either require multiple computing devices or do not support graphical debugging. To address this problem we present GROPG, the first graphical on-phone debugger. We implement GROPG for Android and perform a preliminary evaluation on third-party applications. Our experiments suggest that GROPG can lower the overall debugging time of a comparable text-based on-phone debugger by up to 2/3.**

***Index Terms*—Mobile computing, debugging.**

I. Introduction

Debugging is a common activity during software development and maintenance. For example, in a study of Java developers who were using the Eclipse IDE, over 90% of the developers used the built-in debugger to inspect memory values during program execution [1]. Likewise a recent study found professional software developers to use debuggers heavily for general program comprehension tasks [2].

Debugging mobile phone applications is hard, as current debugging techniques either do not provide powerful debugging features such as a graphical user interface or require multiple computing devices. Specifically, there are currently three types of debugging techniques for mobile phone applications. Mainstream are the first two, which run a standard debugger on a desktop computer. Type (1) techniques attach the debugger to a mobile phone emulator or virtual device that runs the debuggee application. All major mobile application platforms including Android, iOS, and Windows Phone provide such emulators. Emulators are useful but not sufficient, as they do not simulate all phone features precisely. Thus developers resort to type (2) techniques, which attach the desktop debugger to a phone, e.g., via a USB cable. The mainstream techniques therefore ultimately require two connected computing devices, which we also call the *two-device requirement*.

Type (3) techniques were recently pioneered by TouchDevelop [3] and DroidDebugger[1]. These techniques only require a single device, i.e., a mobile phone. However these techniques do not provide the powerful features of desktop-based debuggers. That is, type (3) techniques are either text-based (DroidDebugger) or lack basic debugging features (TouchDevelop). Overall, missing in type (3) techniques is a user interface that allows the programmer to (a) quickly navigate and manipulate the debuggee memory and (b) view the debuggee's current source code location, memory values, call stack, and user interface side-by-side.

[1]https://play.google.com/store/apps/details?id=net.sf.droiddebugger

The two-device requirement of mainstream debugging excludes many, e.g., those who own a smartphone but cannot afford a second computing device such as a desktop computer. This requirement also limits people who have multiple devices, as it constrains debugging style. For example, to get real location sensor readings, a developer has to move the phone, which can be a hassle if the phone is connected to a desktop computer with a short USB cable.

When abandoning mainstream debugging, developers are confronted with the lack of powerful features of on-phone debugging. From the history of debugging we already know that certain debugger features enable higher debugging efficiency. For example, consider the well-known tradeoffs between textual vs. graphical debuggers exemplified by GDB and DDD [4], [5]. We thus argue that mobile phone application debugging overall could be significantly improved by adding powerful features to on-phone debuggers.

The most significant challenge of adding powerful features to on-phone debuggers is limited screen real estate, as dictated by the comparatively small mobile phone screen size. From desktop-based debugging however, developers are used to seeing during debugging several kinds of information on the screen, including the debugged program's current source code location in the context of the surrounding code, the program's runtime memory values, call stack, and user interface.

Besides screen size there are other issues that make it hard to port an existing desktop debugger to mobile phones. Desktop debuggers use user interface elements optimized for keyboard short-cuts and mouse interaction, which do not exist on phones. Similarly, the user interface libraries desktop debuggers are constructed from typically do not exist on phones. For example, Android does not provide the SWT Standard Widget Toolkit on which the mainstream Java and Android debugger Eclipse is built.

Powerful on-phone debugging has only recently come within reach, with mobile phones evolving to powerful interactive computers, narrowing the performance gap with desktop computers in terms of CPU and memory resources. More importantly, mobile phones only recently started to provide high-resolution touch-screens, which enable interactive debugging.

In this paper we present our initial work on providing on-phone debugging with a powerful graphical user interface. That is, we describe the design and implementation of GROPG, the first ***Gr***aphical ***O***n-***P***hone Debu***g***ger. We implement GROPG for Android as Android is a mobile phone platform that is used widely. We also describe a preliminary comparison of our GROPG prototype with the most closely related tool, the text-based on-phone debugger DroidDebugger.

Although we describe our work in terms of Android application debugging, our techniques can also be applied to debugging related languages on related mobile phone platforms such as iOS and Windows Phone. Our implementation is open source and freely available[2].

II. Background on Android Debugging

In this section we provide necessary background information on the Android operating system and its Java-like infrastructure for application debugging.

A. Android Applications Are Signed Java Applications

Android is an open source operating system that is used widely for mobile phones. For example, it is estimated that in the second quarter of 2012 various vendors shipped a total of over 100 million Android phones[3]. Android consists of operating system services, a Linux-based kernel, and the Dalvik virtual machine. Dalvik is conceptually similar to a Java virtual machine and provides similar memory safety.

While Android is mostly implemented in C, almost all Android applications are written in Java. An application is compiled from Java or Java bytecode to the Dalvik Executable (DEX) bytecode language, the language Dalvik can execute. Android runs an application only if it is digitally signed. Developers can compile and sign an application either for release or for debugging. If an application is compiled for release mode, Android prevents it from being debugged.

B. Android's Infrastructure For Application Debugging

For application-level debugging, Dalvik implements two interfaces defined by the Java virtual machine. This makes it easy for existing Java debuggers to connect to Dalvik and control debuggee applications. At a high level, these two interfaces follow the cooperative debugging model, which assumes that the virtual machine is implemented correctly and faithfully provides its debugging infrastructure. Cooperative debugging is standard for application debugging. Specifically, these two interfaces are the Java Debug Interface JDI and the Java Debug Wire Protocol JDWP and are part of the Java Platform Debugger Architecture JPDA.

A debugger can communicate with Dalvik via the Android Debug Bridge ADB, which consists of client, server, and daemon. Each Android device runs a daemon, which may be discovered by the server. The server can be discovered by clients, which are typically started by tools such as a debugger.

Figure 1 illustrates the Android debugging infrastructure in the context of two-device debugging. The Eclipse IDE is connected with an Android application running on a mobile phone via a communication stack consisting of, bottom-up, a USB cable, the Android Debug Bridge ADB, the Java Debug Wire Protocol JDWP, and the Java Debug Interface JDI. Since Android follows the standard Java debugging interfaces JDWP and JDI, the Eclipse Java debugger only needs minimal adaption for Android debugging. This adaption is provided by the Android Development Tools ADT, which direct debugger interactions over the Android Debug Bridge.

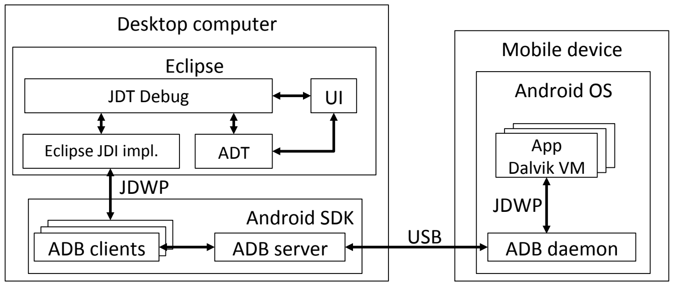


Fig. 1. Architecture overview of a desktop-based debugger, Eclipse, using the Android debugging infrastructure for debugging an application running on a mobile phone.

III. Graphical On-Phone Debugging

The biggest challenge in the design of a graphical on-phone debugger is the design of its user interface, as the user interface has to provide within a mobile phone's constraints the features developers expect from desktop-based debugging.

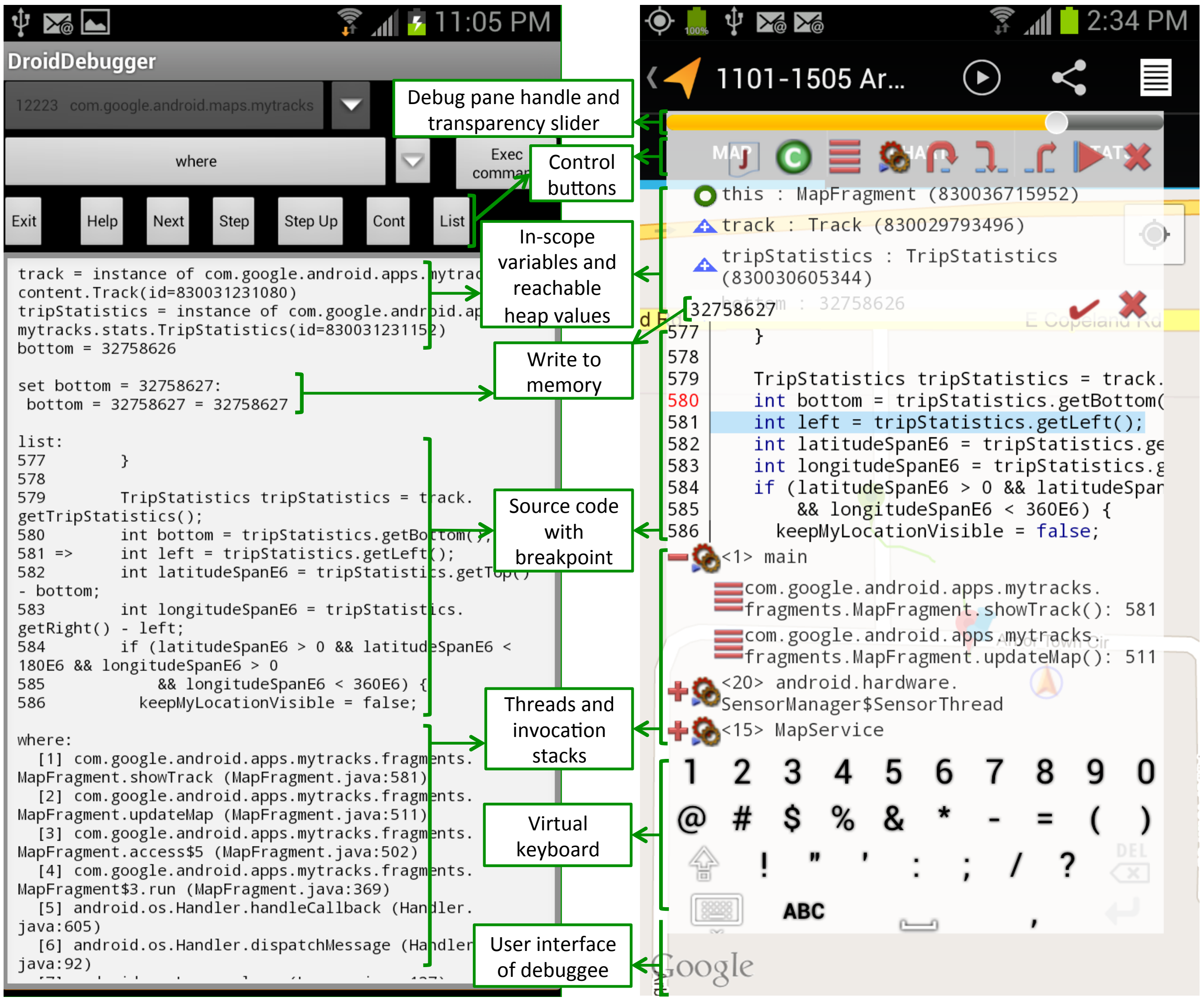
Figure 2 shows the main user interface of our GROPG graphical on-phone debugger on the right and compares it with the closest related tool, the text-based on-phone debugger DroidDebugger. At a high level, GROPG and DroidDebugger make available similar kinds of information.

The main difference between the two approaches is in how they display and let the user interact with debugging information. DroidDebugger provides a text-based user interface, essentially via a shell. The user types command lines and DroidDebugger prints its responses as text output. GROPG on the other hand provides information graphically and interactively. For example, whereas DroidDebugger just prints out the fields of an object, GROPG users can expand such fields and their child fields iteratively by tapping on the desired fields in a graphical display of such a linked data structure. Moreover, whereas DroidDebugger occupies the entire screen, GROPG displays the debugger in a transparent layer on top of the debuggee application.

If the developer wants to interact with the debugged application's user interface, DroidDebugger forces her to frequently switch contexts between debuggee and debugger, as the debugger occupies the entire screen. GROPG on the other hand provides a transparent layer or debug pane, which allows the user to interact with debugger and debuggee on the same screen, saving the user from frequent context switches. The user can move the debug pane by dragging its handle and adjust the pane's transparency with its slider, to provide just

[2]http://cseweb.uta.edu/~tuan/GROPG/

[3]http://www.idc.com/getdoc.jsp?containerId=prUS23638712


11:05 PM
DroidDebugger
12223 com.google.android.maps.mytracks
where
Exec command
Exit
Help
Next
Step
Step Up
Cont
List
2:34 PM
1101-1505 Ar...
Debug pane handle and transparency slider
Control buttons
In-scope variables and reachable heap values
Write to memory
Source code with breakpoint
Threads and invocation stacks
Virtual keyboard
User interface of debuggee
1 2 3 4 5 6 7 8 9 0
@ # $ % & * - = ( )
! " ' : ; / ?
ABC
DEL
Google

```
track = instance of com.google.android.apps.mytrac
content.Track(id=830031231080)
tripStatistics = instance of com.google.android.ap
mytracks.stats.TripStatistics(id=830031231152)
bottom = 32758626

set bottom = 32758627:
 bottom = 32758627 = 32758627

list:
577         }
578
579         TripStatistics tripStatistics = track.
getTripStatistics();
580         int bottom = tripStatistics.getBottom();
581 =>      int left = tripStatistics.getLeft();
582         int latitudeSpanE6 = tripStatistics.getTop()
- bottom;
583         int longitudeSpanE6 = tripStatistics.
getRight() - left;
584         if (latitudeSpanE6 > 0 && latitudeSpanE6 <
180E6 && longitudeSpanE6 > 0
585             && longitudeSpanE6 < 360E6) {
586           keepMyLocationVisible = false;

where:
  [1] com.google.android.apps.mytracks.fragments.
MapFragment.showTrack (MapFragment.java:581)
  [2] com.google.android.apps.mytracks.fragments.
MapFragment.updateMap (MapFragment.java:511)
  [3] com.google.android.apps.mytracks.fragments.
MapFragment.access$5 (MapFragment.java:502)
  [4] com.google.android.apps.mytracks.fragments.
MapFragment$3.run (MapFragment.java:369)
  [5] android.os.Handler.handleCallback (Handler.
java:605)
  [6] android.os.Handler.dispatchMessage (Handler.
java:92)
```

```
this : MapFragment (830036715952)
track : Track (830029793496)
tripStatistics : TripStatistics
(830030605344)
32758627  bottom : 32758626
577    }
578
579    TripStatistics tripStatistics = track.
580    int bottom = tripStatistics.getBottom(
581    int left = tripStatistics.getLeft();
582    int latitudeSpanE6 = tripStatistics.ge
583    int longitudeSpanE6 = tripStatistics.g
584    if (latitudeSpanE6 > 0 && latitudeSpar
585        && longitudeSpanE6 < 360E6) {
586      keepMyLocationVisible = false;
<1> main
   com.google.android.apps.mytracks.
   fragments.MapFragment.showTrack(): 581
   com.google.android.apps.mytracks.
   fragments.MapFragment.updateMap(): 511
<20> android.hardware.
SensorManager$SensorThread
<15> MapService
```

Fig. 2. DroidDebugger (left) and GROPG (right) while debugging the My Tracks application on a Samsung Galaxy S3 phone. In both cases the programmer set a breakpoint at line 580 of the MapFragment class. After My Tracks stopped at the breakpoint, the programmer stepped to line 581, inspected in-scope memory values, and updated the local variable named bottom.

enough visual contrast between debugger and the underlying debuggee user interface.

GROPG enables a workflow that mirrors debugging of a desktop application. That is, the user can load source code files into the debugger, navigate through them, set breakpoints by tapping a line, view and edit the list of all active breakpoints, inspect in-scope memory values, step into, over, and out of instructions, inspect the current threads and runtime stacks, jump to calling methods, and change the values of in-scope memory and heap locations. All actions are available via graphical on-phone interactions, via tap and multi-touch.

IV. GROPG Implementation for Android Phones

Figure 3 shows an overview of the GROPG implementation for Android. We re-use many components of the two-device setup of Figure 1. That is, debugger and Dalvik communicate via ADB and the standard debugging interfaces JDWP and JDI. But instead of a USB cable, our ADB components communicate directly over a network socket. External machines should not connect to this socket and take control of our virtual machine with debug commands. We prevent that scenario by only allowing local connections to this socket. This on-phone communication via a network socket works for Android versions 2.3 to 4.2.1.

To run all components on a single mobile phone, we structure our system as a graphical front-end of the minimal debugger JDB. This setup mirrors the architecture of DroidDebugger, which builds on another JDB Android port. JDB in turn is built on the JDI reference implementation and is one of the example debuggers that are part of JDPA.

Following is the main workflow. After setting a breakpoint and starting the debuggee, the programmer interacts with the

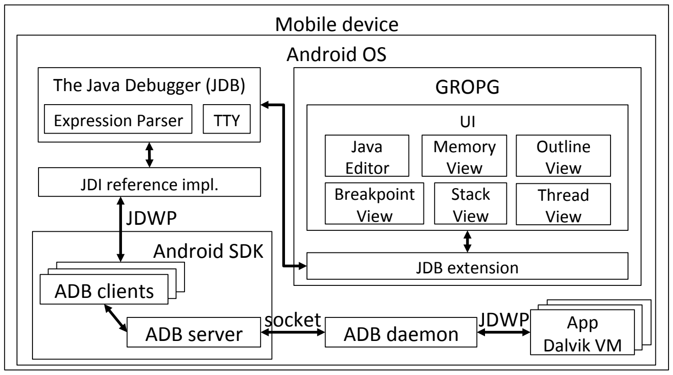


Fig. 3. The graphical on-phone debugger GROPG is built on the same ADB, JDWP, and JDI debugging infrastructure as the two-device debugger of Figure 1.

full debuggee UI. When the debuggee is paused (e.g., at a breakpoint), GROPG displays its debug pane on top of the debuggee UI. After performing debug actions, the programmer switches back to interacting with the full debuggee UI. At this point GROPG saves the current state of the debug pane and removes it from the screen. GROPG will recreate its debug pane UI state at the next debuggee execution break.

V. Preliminary Experience

We compare GROPG with the closest related tool, Droid-Debugger, on a typical debugging task performed on three example open source applications. Table I summarizes the applications and their size. F2C prompts the user for a temperature and converts it from Fahrenheit to Celsius. My Tracks[4] uses a phone's GPS sensor to record the user's motion outdoors. ZXing[5] is a library that enables barcode scanning via a phone's camera. All measurements were taken on a dual-core 1.5GHz, 2GB RAM Samsung Galaxy S3 mobile phone running the recent Android 4.0.4 version.

TABLE I
Memory consumption of application and debugger [MB] and debugging time [min:s]. GROPG has a higher memory footprint but enables faster debugging.

Debuggee	LOC	DroidDebugger			GROPG		
F2C	27	3	10	2:13	3	24	45
My Tracks	21,563	19	11	3:35	19	26	52
ZXing	5,756	7	12	2:53	7	25	1:01

The debugging task we chose for this comparison is to (1) start debugger and debuggee, (2) attach debugger to debuggee, (3) set one breakpoint, (4) once the debuggee reaches the breakpoint step to the next instruction, and (5) display current memory values and the frame stack of the current thread. We set the breakpoints at F2C class F2CActivity line 17, My Tracks class MapFragment line 580 (Figure 2), and ZXing class CaptureActivity line 440. For each task, Table I shows the peak memory consumption and the total time spent.

[4]https://play.google.com/store/apps/details?id=com.google.android.maps.mytracks
[5]https://play.google.com/store/apps/details?id=com.google.zxing.client.android

For space reasons we can break out sub-step measurements only for one step. Table II shows the sub-steps of step (3), setting a breakpoint, for My Tracks. The individual times depend on user experience and will vary among users. But the bottom line is that a GROPG user can set a breakpoint faster and with fewer sub-steps, as she can leverage GROPG's comprehensive debug pane and graphical user interface.

TABLE II
GROPG needs fewer steps for setting a breakpoint.

	DroidDebugger		GROPG	
1	Open source file in external tool	21	Open source file	18
2	Review code in external tool	19	Review code	19
3	Remember class, line for breakpoint	5		
4	Switch back to debugger	3		
5	Recall command syntax	0		
6	Type *stop at com.google.android.apps. mytracks.fragments.MapFragment:580*	63	Tap line	1
7	Tap *Exec command*	1	Tap ● button	1
		112		39

Given the well-known trade-offs between graphical and text-based debugging our measurements are not surprising. GROPG has a higher memory overhead, due to the addition of a graphical front-end. The memory overhead is modest, with some 25MB on a 2GB RAM phone, and does not seem to increase significantly with debuggee size. Table I also suggests that GROPG can reduce debugging time by up to 2/3.

VI. Conclusions and Future Work

We described GROPG, the first graphical on-phone debugger. For future work, we want to integrate our approach with on-phone coding [3], to create a full on-phone IDE. We also want to address many of the traditional debugging challenges on-phone, including cross-language Java and native code debugging [6] and support for why-questions [7].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1017305 and 1117369.

References

[1] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, Jul. 2006.
[2] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proc. 34th ACM/IEEE International Conference on Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 255–265.
[3] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich, "Touchdevelop: Programming cloud-connected mobile devices via touchscreen," in *Proc. 10th SIGPLAN ONWARD*. ACM, 2011, pp. 49–60.
[4] A. Zeller and D. Lütkehaus, "DDD - A free graphical front-end for Unix debuggers," *SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, Jan. 1996.
[5] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, Sep. 2008.
[6] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, "Debug all your code: portable mixed-environment debugging," in *Proc. 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2009, pp. 207–226.
[7] A. J. Ko and B. A. Myers, "Designing the Whyline: A debugging interface for asking questions about program behavior," in *Proc. ACM SIGCHI CHI*. ACM, Apr. 2004, pp. 151–158.