Debugging Cross-Platform Mobile Apps without Tool Break

Christoph Hausmann, Patrick Blitz, Uwe Baumgarten

c.hausmann@weptun.de, p.blitz@weptun.de, baumgaru@in.tum.de

Abstract: Besides its use in the web, the JavaScript programming language has become the basis of some of today's most important mobile cross-platform development tools. To enable and simplify debugging in such environments, this paper presents a novel method for debugging interpreted JavaScript code. The described method uses source code instrumentation to transform existing JavaScript programs in a way that makes them debuggable when executed in any standard JavaScript environment.

1 Introduction

In recent years, the number of applications that runs on smartphone operating systems like iOS or Android, commonly referred to as "apps", has grown tremendously[1]. However, developing such apps is quite complicated, as all common hardware platforms require a different toolset. To simplify this process, several cross-platform development tools have been developed recently. While often solving the issue of developing the business logic and a user interface for multiple platforms in an integrated fashion, no cross-platform development tool available at the time of writing provides a debugging approach which is directly integrated into the IDE. But for a high quality software product, it is important to be able to debug the created code [GHKW08]. Thus, we present a debugger which is integrated into a cross-platform development environment.

In section 2, we examine the existing work in the area of mobile cross-platform development tools. Based on this overview we derive the functional and non-functional requirements a debugger for such environments has to fulfill and compare them to the features of existing JavaScript debuggers. Section 3 describes our debugging approach, as well as the context of our work. The following section 4 shows the implementation of our concept using the existing cross-platform mobile development tool *AppConKit* as a basis. In section 5 we validate our implementation using an independent JavaScript test suite and discuss the results, before we end with a conclusion in section 6.

[1]`http://www.mobilestatistics.com/mobile-statistics/`

2 Existing Work and Requirements

Mobile Development Tools Examining the current landscape of software development tools for app development, we can separate them into five main groups as laid out in [WIM11]:

1. **Native development tools** from the operating system manufactures, e.g. the Android or iOS SDK. They all allow for on-device debugging [And] [iOS].
2. **Cross-compiler tools**. They might also allow for debugging, for example MonoTouch [Mon]. However, these cross-compiler tools do not allow for direct development of apps on multiple platforms - there is still effort involved in porting code between platforms.
3. **Web (HTML5) frameworks** like jQuery or Sencha Touch. These frameworks use a JavaScript debugger in the browser on the developer's PC to debug apps built with them. However, they do not provide an integrated debugger that allows for debugging the created apps directly on a mobile device. [jQu] [Sen]
4. **Hybrid app frameworks**. These combine native and web technology in one app. A common framework in that area, PhoneGap [Pho11], also does not provide means of debugging the apps on the mobile device.
5. **Application description languages** for cross-platform apps. An app created with the most prominent example of app description languages, the Appcelerator platform, cannot be debugged at all [App].

Requirements As long as software is written by humans, they will make mistakes leading to faulty or buggy programs [GHKW08]. These bugs are caused by errors in the source code, in the software design, and sometimes also by the compiler. The activity of analyzing those faults or bugs is called *debugging* [Ros96]. As outlined by Rosenberg ([Ros96]), context is very important for a debugger. To maximize the available context, the debugger should be integrated into the development environment, as all necessary information is available there. This has the added benefit of reducing the impact of tool changes .

The functional requirements for a debugger, according to the book *The Art of Debugging with GDB and DDD* by N. Matloff and P. Salzman, facilitate *The Principle of Confirmation* [MS08, p. 2]. This principle states, that a debugger helps confirming that all assumptions a programmer makes about his program are really true during runtime. To verify these assumptions, most debuggers provide the following set of features:

1. **control** the program flow.
2. **inspect** the contents of variables.
3. **modify** the contents of variables.

Our debugger will be implemented for the JavaScript programming language. JavaScript is a high-level, dynamic and untyped programming language with support for both object-oriented and functional programming styles. It is an interpreted language, which derives its syntax from Java. However, it is not related to Java in any way. [Fla11, p. 1] While JavaScript was made famous for its use as a scripting language for the web, it can be embedded into existing applications to add scripting support to them. On all current mobile operating systems, a powerful JavaScript engine can be run.

Our research shows that the non-functional requirements for an integrated mobile cross-platform debugger are:

- **Integrated User Interface**: The UI of the debugger should be integrated into an existing Integrated Development Environment (IDE) to minimize tool breaks and maximize the available context.
- **Low Response Time**: The UI of the debugger should feel responsive and the delay between firing a command and its response should not be higher than 500 ms on average, based on the Truthful Debugging principle stated in [Ros96].
- **Reliability and Integrity**: It should follow the basic principles of a debugger (Heisenberg Principle, Truthful Debugging, Context is the Torch in a Dark Cave) as described in *How Debuggers Work: Algorithms, Data Structures, and Architecture* [Ros96] .
- **Maintainability and Extensibility**: Its design and protocols should be well defined to allow future extensions.
- **Interoperability**: We can not make assumptions or changes in the runtime environment (JavaScript engine) that will run the code to support debugging. Hence debugging has to be supported with any JavaScript engine.

While the functional requirements are the basic requirements for every modern debugger for high-level languages, the non-functional requirements are specialized for the use case of debugging cross-platform applications.

Existing Debuggers There are several known solutions for debugging JavaScript like the Mozilla Rhino debugger [Rhi], Firebug [Fir] or the Chrome V8 debugger [Chr]. These solutions serve their purpose and have proven to work. However, there is one problem: most of them depend on a specific JavaScript engine. In fact, we found only one solution that provides the full set of debugging features but (at least in theory) does not depend on a specific engine: Crossfire [CB11]. But Crossfire requires that the JavaScript engine running the code implements the Crossfire commands and protocol, which violates the non-functional requirement of **Interoperability**. Hence the current situation is illustrated in figure 2.1. Every JavaScript engine we have looked at comes with its own debugger and defines its own remote debugging protocol.

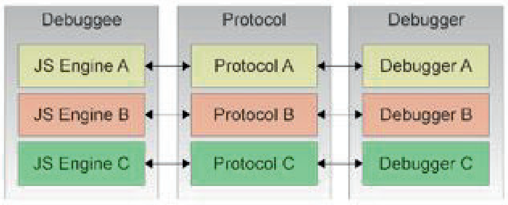

Figure 2.1: Architecture of existing remote debugging solutions for JavaScript.

3 Approach

Proposed Solution As no existing debugger fulfills our requirements, we propose an alternative solution to the challenge of implementing a platform-independent remote debugging solution for JavaScript. We call it the *Universal JavaScript Debugger.*

The basic idea behind the Universal JavaScript Debugger is the following: we know, that there is a JavaScript runtime environment on the debuggee-side[2]. So, if the debuggee-side of the debugging solution itself is implemented using JavaScript, the debugger supports any standard JavaScript engine. This way, we achieve three goals:

1. The solution works with any standard JavaScript engine.
2. There is no need to make changes to an existing JavaScript engine.
3. It can be integrated into an existing IDE.

To achieve this, we modify (instrument) the JavaScript source code before it is sent to the debuggee. As shown in figure 3.1, using that approach, there is only one debugger and only one protocol needed to provide debugging capabilities for various existing JavaScript engines.

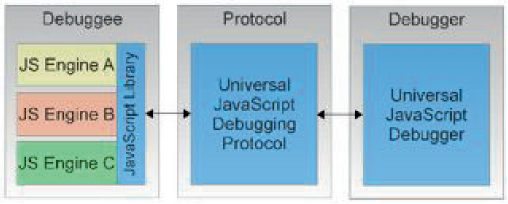

Figure 3.1: Architecture of the Universal JavaScript Debugger.

[2]The program which is being debugged by a debugger is referred to as the debuggee.

Our solution consists of the following parts:

- A **set of source code modification rules** on how to modify an existing JavaScript program to make it debuggable, i.e. to allow for attaching a debugger to the running program and analyzing its state and behavior.
- An **implementation** of those rules using a JavaScript parser and code generator.
- A **JavaScript library** for controlling the program flow and communicating with a remote debug UI.
- A **debug UI** based on the Eclipse software development platform for controlling the debuggee which will be integrated into the *AppConDeveloper* IDE.

Context The concept and implementation of our debugger is based on the AppConKit, a commercial cross-platform native app development tool created by Weptun[3]. We extended it to allow for debugging directly on the mobile device.

The AppConKit is a framework dedicated to the design and implementation of native mobile business applications for touchphones and tablets. It currently supports the iOS operating system from Apple (iPhone and iPad), as well as the Android platform developed by the Open Handset Alliance, where Google is one of the main contributors.

Supported by a graphical interface, mobile application developers are able to create native multi-platform mobile applications using a set of ready-to-use software components provided by the AppConKit. The AppConKit is based on a client-server architecture consisting of the *AppConClient* and the *AppConDeveloper* (or the *AppConServer* after deployment). During development of a mobile application, the AppConClient runs on the mobile device, while the AppConDeveloper runs on the developer's machine.

The challenge of developing applications for heterogeneous platforms and devices is solved by using an abstract platform-independent application description language. This application description, which is directly and automatically generated from the graphical representation of the application, is then interpreted by a special runtime on the mobile device.

In the most recent iteration, the AppConKit has been extended by adding the possibility to express the app's business logic using JavaScript code which is directly executed on the device. The architecture of the AppConKit can be seen in figure 3.2.

[3]`http://www.weptun.de`

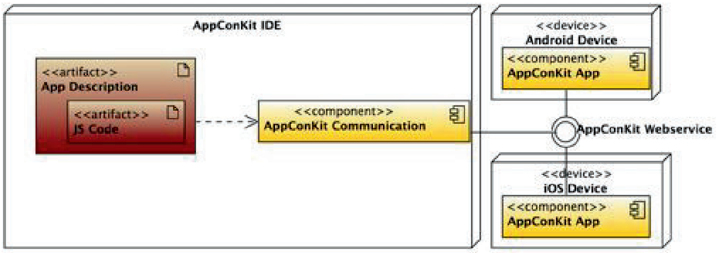

Figure 3.2: Architecture of the AppConKit.

4 Implementation

To evaluate the proposed solution, we implemented a prototypical system and tested it to show that we meet the goals outlined above.

System Architecture The abstract system architecture of the Universal JavaScript Debugger is shown in figure 4.1. On the debuggee-side there is a JavaScript engine which executes the previously instrumented JavaScript code. This instrumented code communicates with a JavaScript debug library, that also runs on the debuggee-side. This library exchanges information with a remote JavaScript debug connector located on the debugger-side. The connector is attached to a controller which is operated by the UI. On the debugger-side there is also the original unmodified JavaScript code written by the developer. It is needed to visualize the current state of the program, i.e. the lines where breakpoints are defined, and the current position where the program is suspended.

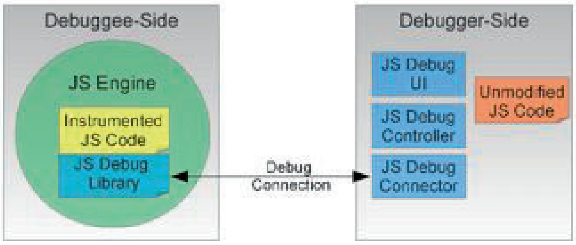

Figure 4.1: Abstract system architecture of the Universal JavaScript Debugger.

The actual system architecture can be seen in figure 4.2. The integration of the Universal JavaScript Debugger into the AppConDeveloper is carried out by extending the Eclipse JSDT [Ecl] environment.

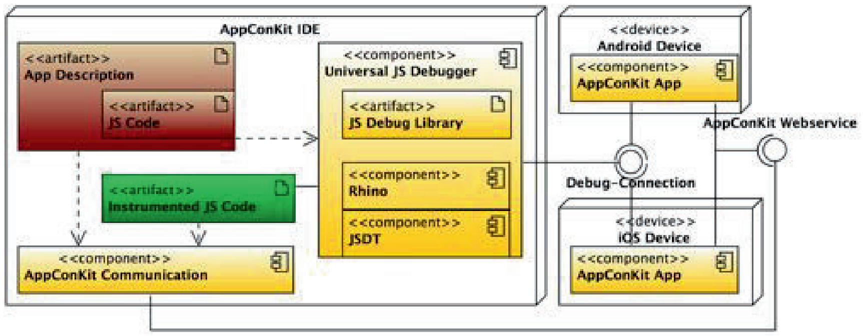

Figure 4.2: System architecture of the AppConKit including the integrated JavaScript debugger.

Integration into the AppConKit The AppConKit development environment can be used in either *Debug* or *Normal* mode. Our debugging procedure is only activated in debug mode.

When using the AppConKit in debug mode, the resulting design as shown in figure 4.2 is basically achieved by combining the designs of figure 3.2 and figure 4.1. We start with the AppConDeveloper, the Eclipse-based development environment used to implement the mobile applications. With the AppConDeveloper, the business logic can be implemented using JavaScript and attached to the mobile application, so that it can be executed on the AppConClient. However, before the app description is sent to the client, the JavaScript code is processed and instrumented as shown below. After that it is embedded into the app description, along with a JavaScript debug library. Upon interpretation of the app description on the debuggee-side, the embedded JavaScript code is extracted and executed within a JavaScript engine. During execution, the JavaScript debug library communicates with the Universal JavaScript Debugger within the remote AppConDeveloper via the debug connection. The AppConDeveloper now has the role of the debugger.

The actual source code modifications were implemented with a JavaScript parser, which creates an Abstract Syntax Tree (AST) of the program. In the next step, the AST is traversed, and calls into a debug library are inserted. Additionally, in order to allow for more sophisticated debugging features, certain patterns in the AST are identified and transformed according to a set of rules. In the final step, the modified AST is transformed into instrumented JavaScript source code by a code generator. We use the Mozilla Rhino environment [Moz] as the JavaScript parser and source code generator.

Instrumentation and Code Rewriting Rules We stated that we use source code instrumentation and other modifications to add additional statements to the source code. In this section, we will highlight the rewriting rules used to modify the code for instrumentation. In the following, we will refer to the original non-modified script as *original script*, and to the modified script as *instrumented script*.

Using source code instrumentation and other modifications of the original script, we want to achieve two goals:

1. The instrumented script gathers additional information about its state at runtime. This information then can be transferred to a remote debugger.
2. The instrumented script can be controlled by a remote debugger. This enables the implementation of breakpoints and step-by-step execution.

In order to reach these goals, we must be able to monitor and alter the behavior of the original script at a very fine-grained level. So the purpose of the following source code modifications is to be able to insert additional function calls at arbitrary locations in the original script. These additional function calls invoke functions of our JavaScript debug library, which are used to gather runtime information, and also to change the original script's behavior. To achieve this, we need to break up complex constructs in the original script into a set of more or less atomic operations. Between these atomic operations, additional statements have to be inserted, which capture the program state before and after those operations.

The two most important statements we insert are the inclusion of the debug library and the debug() statement itself. The debug library is used to manage the gathered runtime information and to communicate with the remote debugger. So the basic step of the source code modification consists of inserting the debug library at the beginning of the original script, which makes it globally available. The functions provided by the library are:

- **debug()** - used by the instrumented script in order to provide runtime information to the debug library. Additionally, the debug library is able to pause execution of the instrumented script by not returning from this function.
- **push()** - used to create a new stack frame and save the associated local variables whenever a new function is called.
- **pop()** - used to remove the top most stack frame and inserted whenever a return statement is found.

Now that the debug library is available, we need to insert a *debug()* call before every executed statement. Events can occur at any statement, and as such we must be able to pause execution at any statement. The formal definition of this transformation rule is very simple, it is shown in table 1.

Source Pattern	**Transformation Rule**
Statement	**debug(...);** *Statement*

Table 1: Source pattern and transformation rule for *Inserting Debug Statements* rule.

In total, we created 7 rules to cover all atomic operations listed in table 2. With the total source code instrumentation rule set we were able to cover every use case outlined there.

5 Validation

Debugger Tests We created test cases to verify that each of the debugger's functions works with every JavaScript source construct. Table 2 shows the test matrix, as well as the final test results. For each row in the test matrix, we implemented a JavaScript function, which uses the JavaScript source construct to be tested. After that, we executed this function six times, each time using a different debugger functionality (*Step In*, *Step Over*, *Modify Variable*, ...).

The test cases were executed using both iOS and Android as a debuggee and with the AppConDeveloper as the debugger. This ensures that no platform-specific errors exist, and that the functional requirements are fulfilled. Additionally, this way also the non-functional requirements **Integrated User Interface** and **Interoperability** can be validated.

			Debugger Functionality					
			Set Breakpoint	Step In	Step Over	Step Out	Inspect Variable	Modify Variable
JavaScript Source Construct	Choice:	If Else	✓	✓	✓	✓	✓	✓
		Switch Case	✓	✓	✓	✓	✓	✓
	Loops:	While	✓	✓	✓	✓	✓	✓
		Do While	✓	✓	✓	✓	✓	✓
		For	✓	✓	✓	✓	✓	✓
		For In	✓	✓	✓	✓	✓	✓
		For Each In	✓	✓	✓	✓	✓	✓
	Special Statements:	Break	✓	✓	✓	✓	✓	✓
		Continue	✓	✓	✓	✓	✓	✓
		Return	✓	✓	✓	✓	✓	✓
		Throw	✓	✓	✓	✓	✓	✓
		With	✓	✓	✓	✓	✓	✓
	Exceptions:	Try Catch	✓	✓	✓	✓	✓	✓
	Functions:	Definition	✓	✓	✓	✓	✓	✓
		Function Call	✓	✓	✓	✓	✓	✓
		Nested Function Calls	✓	✓	✓	✓	✓	✓
	Variables:	Declaration	✓	✓	✓	✓	✓	✓
		Assignment	✓	✓	✓	✓	✓	✓
	Data Types:	Number	✓	✓	✓	✓	✓	✓
		String	✓	✓	✓	✓	✓	✓
		Boolean	✓	✓	✓	✓	✓	✓
		Object	✓	✓	✓	✓	✓	✓
		Array	✓	✓	✓	✓	✓	✓
		Function	✓	✓	✓	✓	✓	✓
		null	✓	✓	✓	✓	✓	✓
		undefined	✓	✓	✓	✓	✓	✓

Table 2: Test matrix and results of debugger related test cases.

Mozilla JavaScript Test Library To verify that the behavior of the original scripts is not changed by our source code modifications and hence to validate the non-functional requirement **Reliability and Integrity**, we used the *Mozilla JavaScript Test Library*[4]. This test suite was created to test the functionality of the core JavaScript engine. It currently consists of 5216 test cases which verify the engine's behavior when executing a JavaScript program. The test cases itself are written in JavaScript, and they are executed using the Java Test Framework *JUnit*[5] and Mozilla Rhino as a JavaScript engine. There is a common JavaScript library used in every test case which provides the basic test infrastructure, e.g. means of comparing the expected and actual behavior, and raising an exception, if the test case has failed. The test cases are executed sequentially and upon completion, a test report is generated, which contains a list of all executed test cases with their status. There are three states:

1. **Passed** - expected and actual behavior are the same, everything works.
2. **Failed** - actual behavior differs from expected behavior, something is wrong.
3. **Error** - there was an error when executing the test case, e.g. a Java exception raised by the Mozilla Rhino JavaScript engine.

To be able to use this large test suite to verify the correctness of our code instrumentation rules, we modified it in the following way:

- When a JavaScript test case is read from file, we process the contents of the file first using our code instrumentation implementation, before handing it over to the test executor. This way, all executed code is instrumented JavaScript code.
- As instrumenting the code takes a few seconds, running the whole test suite sequentially can add up to a few hours in total. In order to verify the correctness of changes to the instrumentation code faster and more often, we further modified the test suite, so that several test cases are run in parallel. On a quad-core machine with 8 parallel threads (using hyper-threading) we could reduce the time for running the complete test suite to less than 45 minutes, which turned out to be an acceptable delay.

To make sure to only discover actual errors within the code instrumentation rules and implementation, we defined a three tiered test methodology, which allows us to ignore errors within third-party components, i.e. the JavaScript engine, or the JavaScript parser.

1. Run: execute the original test cases. This reveals all errors within the JavaScript engine itself. (Test cases: 5216, Passed: 96.3 %, Failed: 3.7 %, Error: 0 %)
2. Run: leave out the failed test cases of the first run. Before executing a test case, parse the JavaScript code, but don't modify the resulting AST. Instead, just generate JavaScript code again and run it. This reveals all errors within the parser and the code generator we use. (Remaining test cases: 5022, Passed: 90.6 %, Failed: 9.4 %, Error: 0 %)

[4]`http://www-archive.mozilla.org/js/tests/library.html`
[5]`http://www.junit.org/`

3. Run: leave out the failed test cases of the second run. For each remaining test case, instrument the original test case, and run the instrumented test case using the test executor. Now we know exactly which test cases failed due to the source code instrumentation, and not because of other errors. (Remaining test cases: 4548, Passed: 97.2 %, Failed: 1.7 %, Error: 1.1 %)

Test Results Figure 5.1 shows the results of the three runs in one chart.

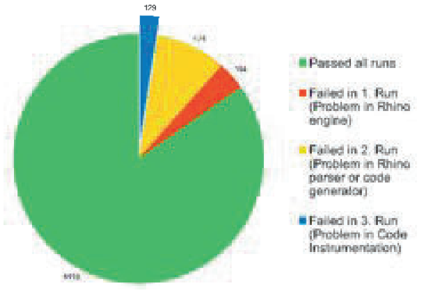

Figure 5.1: Test results achieved using the Mozilla JavaScript Test Library.

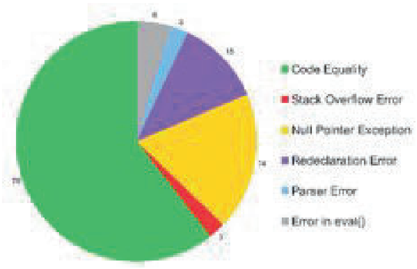

Figure 5.2: Detailed analysis of the test cases which failed due to code instrumentation.

Most of the errors are caused by either the Rhino engine itself, or Rhino's JavaScript parser, which we use. A further analysis of the failures revealed that most of the failures are caused by one of the following reasons:

- Missing dependencies: The test suite contains test cases designed to test functionality only available in a browser, such as means of modifying the HTML document currently displayed in the browser. As these features are not part of a standalone Rhino instance, all these test cases fail. This is the cause of most of the 194 failures by the Rhino engine.
- Code equality checks: In JavaScript, functions can be transformed into a String representation which contains the source code of that particular function. A lot of test cases make use of that feature to check whether the properties of that function match the expected content. These checks are very specific. In fact, it is enough to add an additional whitespace in order for such a test to fail. By using a parser and a code generator, the resulting source code is slightly different than the original one, which is the reason why using the Rhino parser alone is enough to make 474 tests fail. So these test cases can be considered invalid test cases, as they will also fail if there aren't any real problems in the source code.

With that knowledge in mind, we further analyzed the remaining 129 failures which were caused by the instrumented scripts. These are the failures we are interested in, because they could be caused by incorrect code instrumentation rules. For our analysis, we assigned the failures to different categories, as figure 5.2 shows.

- 78 test cases have failed due to code equality checks. These test cases will never work, as by using source code instrumentation, the source code will always be different than what is expected by the test case.
- 27 test cases have failed due to errors within the JavaScript engine itself: 24 Null Pointer Exceptions and 3 Stack Overflow Errors within Rhino's Java code. Until now, we haven't identified the exact cause of these errors in Rhino's source code, however, they are likely to be fixed in future versions.
- 15 test cases failed due to a *Redeclaration Error* raised by the JavaScript engine. This turned out to be a bug in Rhino's implementation of the *eval()* function. We filed a bug report[6] for this error, and there is already a fix available.
- 6 test cases failed due to an incorrect behavior of the *eval()* function in some cases. There seems to be a problem with global objects defined within an *eval()* call. We are still investigating this problem, however, it seems to be an error within Rhino's *eval()* implementation, just like the *Redeclaration Error*.
- 3 test cases failed due to an incorrect handling of a reserved word by Rhino's JavaScript parser.

We were able to either solve every problem which resulted in a failed test case, or to trace back the problem to a third party component. Table 3 contains a summary of the final test results. It shows, that all remaining failing tests are caused by one of the following:

- Problems in the JavaScript engine.
- Problems in the JavaScript parser.
- The test procedure, e.g. by test cases which rely on completely unmodified source code.

Number of Test Cases	**Percentage**	**Result**	**Reason**
4419	84.7 %	Passed	No problems
552	10.6 %	Failed	Invalid test cases: code equality checks
194	3.7 %	Failed	Invalid test cases: missing dependencies
51	1 %	Failed / Error	Problem in Rhino JavaScript engine

Table 3: Summary of the test results achieved using the Mozilla JavaScript Test Library and the Mozilla Rhino JavaScript engine.

[6] `https://bugzilla.mozilla.org/show_bug.cgi?id=784358`

With these results, we can show that our source code modification rules are correct in the sense that they preserve the functionality of the instrumented scripts. And given the sheer number of successful tests, our implemented solution performs well enough to be used in nearly all practical settings. This means that we have reached the non-functional requirement **Reliability and Integrity**.

6 Conclusion

In this paper, we presented a method for device-independent integrated debugging of JavaScript programs which is based on a mobile cross-platform app development framework. We elicited the important functional and non-functional requirements our solution has to fulfill. Based on these requirements, we designed a universal JavaScript debugger and integrated a prototypical implementation into the AppConDeveloper IDE. Using this solution, we have run extended functional tests to ensure that we meet all requirements. We also were able to show that our approach could be integrated into an existing IDE and be used to debug code running on a remote device in the same context as writing or running it. With this prototypical solution, we have proven that the approach is generally feasible.

However, our work is not finished yet. We have started to evaluate the system in real-life scenarios and plan to release it to a wider public in the near future. In this context, several other studies will be carried out to lead to the final goal of simplifying the development of integrated mobile apps in the same way that IDEs [KKNS84] did for normal application development.

References

[And] Debugging | Android Developers. http://developer.android.com/tools/debugging/index.html. [Online; accessed 2013-02-07].

[App] Appcelerator: JavaScript Debugging. http://developer.appcelerator.com/question/27731/javascript-debugging. [Online; accessed 2013-02-07].

[CB11] Michael G Collins and John J Barton. Crossfire: multiprocess, cross-browser, open-web debugging protocol. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 115–124, New York, NY, USA, 2011. ACM.

[Chr] Chrome Dev Tools Debugger Features. http://code.google.com/p/chromedevtools/wiki/EclipseDebuggerFeatures. [Online; accessed 2012-10-14].

[Ecl] JavaScript Development Tools (JSDT). http://www.eclipse.org/webtools/jsdt/. [Online; accessed 2012-08-29].

[Fir] JavaScript Debugger and Profiler : Firebug. https://getfirebug.com/javascript. [Online; accessed 2012-10-14].

[Fla11] David Flanagan. *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media, 2011.

[GHKW08] Thorsten Grötker, Ulrich Holtmann, Holger Keding, and Markus Wloka. *The Developer's Guide to Debugging (Google eBook)*. Springer, 2008.

[iOS] Xcode User Guide: Debug and Tune Your App. http://developer.apple.com/library/ios/#documentation/ToolsLanguages/Conceptual/Xcode_User_Guide/060-Debug_and_Tune_Your_App/debug_app.html. [Online; accessed 2013-02-07].

[jQu] How to Debug Your jQuery Code. http://msdn.microsoft.com/en-us/magazine/ee819093.aspx. [Online; accessed 2013-02-07].

[KKNS84] Benn R. Konsynski, Jeffrey E. Kottemann, Jay F. Nunamaker, and Jack W. Stott. PLEXSYS-84: An Integrated Development Environment for Information Systems. *Journal of Management Information Systems*, 1(3):64–104, 1984.

[Mon] Debugging MonoTouch. http://docs.xamarin.com/ios/Guides/Deployment%252c_Testing%252c_and_Metrics/Debugging_in_MonoTouch. [Online; accessed 2013-02-07].

[Moz] Rhino overview | Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Rhino/Overview. [Online; accessed 2012-08-29].

[MS08] Norman Matloff and P J Salzman. *The Art of Debugging with GDB and DDD*. No Starch Press, 2008.

[Pho11] Debugging PhoneGap. http://phonegap.com/2011/05/18/debugging-phonegap-javascript/, 05 2011. [Online; accessed 2013-01-03].

[Rhi] Rhino Debugger Features. https://developer.mozilla.org/en-US/docs/Rhino/Debugger. [Online; accessed 2012-10-14].

[Ros96] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley, 1996.

[Sen] How to Debug Sencha Touch 2 Apps. http://stackoverflow.com/questions/10127244/how-to-debug-sencha-touch-2-apps. [Online; accessed 2013-02-07].

[WIM11] Felix Willnecker, Damir Ismailovic, and Wolfgang Maison. Architekturen mobiler Multiplattform-Apps. In Stephan Verclas and Claudia Linnhoff-Popien, editors, *Smart Mobile Apps*, number 26 in Xpert.press. Springer DE, 2011.