RWTH AACHEN UNIVERSITY

Utilization and Visualization of Program State as Input Data in a Live Coding Environment

Diploma Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Ewgenij Belzmann

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Bernhard Rumpe

Registration date: October 1st, 2012
Submission date: April 30th, 2013

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, April 2013
Ewgenij Belzmann

Contents

List of Figures

List of Tables

Abstract

Live coding seeks to break the *code-compile-test loop* familiar to many developers by executing code in the background and presenting the result to the developer without distracting him or her from coding. Many such applications and prototypes have been created, with different interaction and visualization solutions.

This thesis aims to provide a technical basis for a live coding environment by offering means to execute code in the background while the developer is still editing it, and to provide the changes in program state. This allows to see the flow of data through the course of the application while writing code, and to locate possible errors and bugs more easily. Our approach is not tied to any particular visualization and allows to explore the advantages and disadvantages of different visualization solutions in the future without being limited by technical limitations.

The program presented in this thesis is written in JavaScript as a server for the framework Node.js. It can run Node.js programs and sends back the changing program states to the calling application. It is primarily written to work with the Brackets code editor, but can in principle be called from any JavaScript editor that is capable to communicate via the *WebSocket* protocol.

Überblick

Live coding versucht, den Kreis *code-compile-execute*, der vielen Entwicklern bekannt ist, zu brechen, indem es den Code im Hintergrund ausführt und dem Entwickler das Ergebnis zeigt, ohne sie oder ihn vom Programmieren abzulenken. Viele solche Anwendungen und Prototypen wurden bereits entwickelt, mit verschiedenen Interaktions- und Visualisierungslösungen.

Diese Diplomarbeit strebt an, eine technische Basis für eine Live-Coding-Umgebung bereitzustellen, indem es sie Mittel anbietet, Code im Hintergrund auszuführen noch während der Entwickler diesen bearbeitet, und die Veränderungen im Programmzustand anzuzeigen. Dies erlaubt es, den Datenfluss in der Programmausführung während der Bearbeitung zu sehen und Fehler und Bugs leichter zu lokalisieren. Unser Ansatz ist nicht an irgendeine bestimmte Visualisierung gebunden und erlaubt es in Zukunft, Vor- und Nachteile verschiedener Visualisierungslösungen zu erforschen, ohne dabei von technischen Beschränkungen gebunden zu sein.

Das Programm, das in dieser Arbeit vorgestellt wird, ist in JavaScript als ein Server für das Node.js-Framework geschrieben. Es kann Node.js-Programme ausführen und sendet die wechselnden Programmzustände zurück an die aufrufende Anwendung. Es ist in erster Linie geschrieben, um mit dem Brackets-Code-Editor zusammenzuarbeiten, aber im Prinzip kann es aus jedem JavaScript-Editor aufgerufen werden, der in der Lage ist, mit Hilfe des *WebSocket*-Protokolls zu kommunizieren.

Acknowledgements

First of all, I want to thank Prof. Borchers for getting me interested in HCI in a single lecture. I since have become increasingly aware of bad interfaces and annoy all my friends and family with it. I also am very grateful to Jan-Peter Krämer for taking me on as a diploma thesis student and offering me a very interesting topic that has taught me a lot, and giving me countless advice during the my thesis. And another thanks goes to Leonhard Lichtschlag for introducing my to Jan-Peter, by literally dragging me into his office unannounced. And thanks to Joachim Kurz for proving me a lot of feedback on my program and thesis.

I also want to thank my parents and grandparents for supporting and educating my every step of my studies since the very early childhood, and my girlfriend Julia Zaloga for being very supportive and patient for the last 6 years.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

```
myClass
```

The whole thesis is written in American English.

Download links are set off in coloured boxes.

File: myFile[a]

[a] http://hci.rwth-aachen.de/public/folder/file_number.file

Chapter 1

Introduction

Development happens in edit-compile-test loop

Most traditional integrated development environments (IDEs) presuppose a particular mode of interaction during the development of program features or debugging. The code is written in a text editor (maybe with assistance of some graphic tools), then it has to be compiled (unless the language in question is interpreted during execution or just-in-time compiled (JIT-compilation)), and then it has to be executed to see the changes in the program. The debugging stage often comes with yet another separate interface. This is known as the *edit-compile-test loop*.

Larger ignorance time leads to a larger fix time

The *compile*-stage of the loop does not constitute *productive work* by the developer, and the test might also contain steps that take time away from actual development. So the developer often tries to increase the time spent on productive work by compiling and testing less often. This, unfortunately, leads to a larger gap between a possible introduction of an error into the code and its discovery by the programmer. This gap is referred to as *ignorance time*. Studies by Saff and Ernst [2003, 2004] have shown that larger ignorance times lead to a larger *fix time*.

Definition: *Ignorance time*

IGNORANCE TIME:
Time between the introduction of an error and the programmer becoming aware of it

Definition: *Fix time*

FIX TIME:
Time between the programmer becoming aware of an error and him fixing it

Continuous compilation shows static code errors

Attempts to shorten this gap by providing the developer immediate (or at least fast) feedback without disturbing his work have been made for a long time. Schwartz et al. [1984] have developed an IDE for Pascal that incrementally compiled the program in the background and showed static code errors in the editor. The personal computers of the time were not powerful enough to work with the IDE productively, but similar technology has since been integrated in many popular IDEs (Eclipse[1], Visual Studio[2] and others) and is commonly known as *continuous compilation*.

Live coding *executes* code in the background and shows also runtime errors

Continuous compilation is only able to show static errors, runtime errors can only be detected by the developer actually running the program. So the next logical step is *continuous execution*. This means that code is executed in the background during development, and the output from the code is made visible to the user. This programming method is called *live coding*.

Background execution often tightly bound to presentation

Most live coding solutions presented in academic literature (often they are prototypes used for presentation or evaluation) or even available for use have a fairly tight coupling between the runtime, that is the part of the program executing or evaluating the code or parts of it, and the visualization components. In some cases, like Rehearse by Choi et al. [2008], Choi [2008], they even enforce a specific model of interaction during coding. This unfortunately means that, to test or evaluate a new type of visualization or interaction, one often has to re-implement not only the frontend components but also the backend responsible for the evaluation of the code.

Client-server architecture allows to implement visualization separately from a stable backend

Instead, we propose a server-client architecture with the server being responsible for executing the code and sending the data to the client (or potentially several clients) and the client doing the visualization as it sees fit. This server

[1] http://www.eclipse.org
[2] http://www.microsoft.com/visualstudio

can be used as a robust basis for different visualization solutions without burdening the developer with the details of the language or the execution environment.

This thesis first present the available research on the topic (Chapter 2), gives an overview of some aspects of JavaScript (Chapter 3), and then presents the details of our implementation (Chapter 4). Lastly, we evaluate the presented solution (Chapter 5) and discuss possibilities for future work in this field (Chapter 6).

Chapter 2

Related Work

Magpie editor introduced continuous compilation

There have been many attempts at shortening the time span between the *fix time* by shortening the *ignorance time*. Some of the earliest works in this area, e.g. the *Magpie* programming environment by Schwartz et al. [1984], have been concerned with incremental compilation of the edited code, providing almost immediate feedback on possible syntax and compile time errors. Since the power of even high-end workstations of the time was fairly limited, *Magpie* does not recompile the whole program on every change, instead breaking it up in fragments natural to the programming language in use, *Pascal*, and performs analysis in those fragments separately. This method of compilation has since been integrated in many software development tools (e.g. Eclipse[1]) and is commonly known as *continuous compilation* (as apposed to the term *incremental compilation* used in [Schwartz et al., 1984]). (One must take care, though, to specify more precisely what is meant by continuous compilation, since that term has also been used with a different meaning, for example, in [Childers et al., 2003], denoting a way for dynamically optimizing code at runtime by recompiling and optimizing parts of it.

[1] http://www.eclipse.org

Running unit tests in the background helps prevent regression errors

A study on the efficacy of reducing the time between the introduction and the discovery of an error has been conducted by Saff and Ernst [2003]. They were primarily concerned with *regression errors* and a test-first approach to development. The proposed solution was to continuously run unit tests in the background and to present the results to the developer (instead of running those tests at the developers command, as is common). A user study conducted by Saff and Ernst [2003] has shown a correlation between the *ignorance time* and the *fix time*, a result that has been supported by a further study ([Saff and Ernst, 2004]). Incidentally, the second study also served as an empirical proof for the benefits of continuous compilation, by showing a doubled likelihood for completing the given task on time with it, with even more participants completing it with the addition of continuous testing. Unfortunately, the tested approach only covers cases in which there not only exists a full specification of the desired behavior but also a test suite to test those requirements, or at the very least a test suite covering the past correct behavior of the program that is to be preserved; the test subjects in both studies have been provided with complete test suites required for testing.

Definition: *Regression error*

REGRESSION ERROR:
An error in the behavior of the program that has not been present in a previous version and has been accidentally introduced into the code.

Rehearse allows interactive coding with separate interface

The *Rehearse* editor by Choi et al. [2008], Choi [2008] evaluated another form of live coding interaction. It is called *interactive development* and allows to write code line by line with the program adding the output for the last written statement to the editor window after it has been typed (see Figure 2.1). It also allows to undo previously executed statements and to return to a previous program state. The studies showed that, despite benefits from immediate feedback, the users were confused from separate interfaces for classical and interactive coding. This shows that any type of live coding application has to be integrated with the familiar programming tools as much as possible.

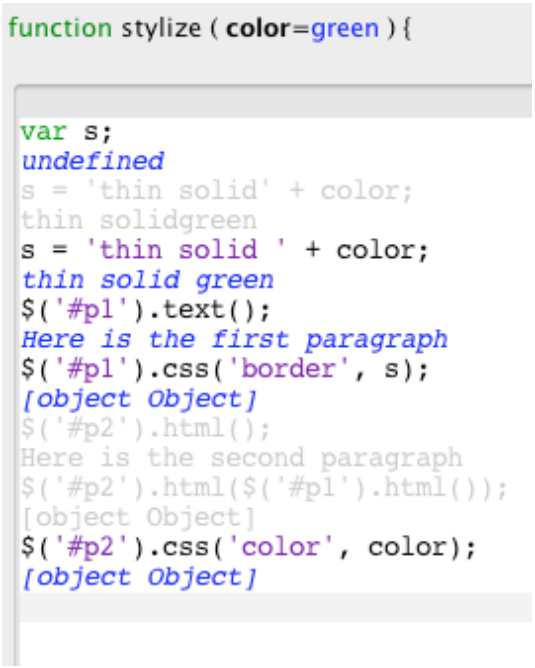

Figure 2.1: A screenshot of the interactive editor in Rehearse. Blue lines show the feedback of the statement above them. Grey lines are statements that have been undone.

Example centric programming by instrumenting execution environment

Another project that uses instrumentation and background execution in a separate environment to explore program state is an Eclipse plugin by Edwards [2004]. They primarily propose the use of concrete examples of executed code (i.e., code executed with specific parameters or example values) to lower the level of abstraction needed during development. The approach is called *example centric programming*. They also explore the possibilities of this technique for related fields: example centric teaching, debugging and testing. The program is also used for a form of test-driven development, by defining stubs for functions and a form of assertions that are bound not to a point in code but to a step in the program execution, and gradually shaping out the code until the assertions are passed. The main technical difference between the aforementioned software and ours is the approach to instrumentation: here the instrumentation is applied to the execution environment itself (BeanShell, a Java interpreter), while we instrument the code itself and execute it in a built-in virtual machine in Node.js.

Chapter 3

JavaScript

We present some unexpected features of JavaScript

Since our application deals with the instrumentation and execution of JavaScript code (while also being implemented completely in JavaScript), we want to give a short introduction into the language before going into details of our solution. The scope of the thesis does not allow us to give a detailed description of the whole language (for a more complete depiction see, for example, [Crockford, 2008]). We instead will concentrate on those parts of the language that are important to our solution and may appear unexpected or unusual to the reader. The chapter assumes that the reader is familiar with C-like languages and the concepts of object-oriented and functional programming.

JavaScript is dynamic, weakly typed, imperative, functional and object-oriented

JavaScript is a *dynamic, weakly typed, multi-paradigm* language. Its syntax is based on C and similar languages, and most constructs will be familiar to anyone with experience with any C-style language. But semantically JavaScript is more related to functional programming languages like Lisp and Scheme ([Crockford, 2008]). Functions are first-class objects; they can be assigned to any variable, and used as parameters in other functions. JavaScript is also *object-oriented*, but it has no notion of classes. Instead it is *prototype-based* (objects can inherit methods and properties from other existing objects).

3.1 Scope

JavaScript has *lexical scoping*

Scope refers to the context in a program in which an identifier is valid. Like many other C-like languages, JavaScript has *lexical scoping*. This means that the scope of an identifier is determined by its position in the source code, and not by the execution context (like in dynamic scoping). Consider the following example:

```
1  function a() {
2    ...
3  }
4
5  function b() {
6    var c = 5;
7
8    function d() {
9      ...
10   }
11
12   a();
13   d();
14 }
```

Function `b` declares a variable `c` and an inner function `d`. It then calls the functions `a` and `d`. Because function `a` is declared outside of `b`, it has no access to `c`. Function `d` on the other hand has access to `c`, since it was declared in its scope.

3.1.1 Function Scope and Hoisting

JavaScript has *function scope*

What separates JavaScript from most other languages in the C family is the fact that it does not have *block scope*. Instead it uses *function scope*. This means that an identifier declared anywhere in a function is valid throughout the function, irrespective of the block it was declared in. Unlike Pascal, for example, which also has function scope, but only allows to declare local variables and inner functions in a function's header, JavaScript permits declarations (almost) anywhere in the body of functions.

Most surprising with regard to function scope is that a variable or function may be valid seemingly before it was actually declared. See the following example:

Variable and function declaration are moved to the top of the context

```
1 var a = 1;
2
3 function b() {
4     a = 2;
5     var a;
6 }
7
8 b();
```

The assignment to the variable *a* in line 4 affects *not* the global variable declared in line 1, but the local variable declared in line 5. Actually the variable declaration gets "moved" up to the beginning of the function *b()* and is already present at the point of the assignment. This mechanism is referred to as *hoisting*. A possible initialization of the variable is not hoisted with the declaration. Thus the following code

```
1 var a = [1, 2, 3];
2
3 function b() {
4     console.log(a[0]);
5     var a = [4, 5, 6];
6 }
7
8 b();
```

would print to the console neither a 0, nor a 4. The access to variable `a` in line 4 refers, as explained previously, to the local variable declared in line 5. But at this point it is not yet initialized, and an attempt to access a non-existent property will return the special value `undefined`.

3.1.2 Closure

Another aspect of JavaScript that is tied in with the issue of scope is *closure*. Closure is the local scope of a function that is available to all function declared in it (inner functions), even after the outer function has returned. See the following code for an example:

Functions have access to their outer scope at all times

```
1  function outer(arg) {
2    function inner() {
3      console.log(arg);
4    }
5
6    return inner;
7  }
8
9  var func = outer("test");
10 func();
```

Here, the variable `func` is initialized with the return value of the function `outer` with the argument `"test"`. `outer` returns the function `inner` that prints the value of `arg` to the console. After the function `outer` returns, the variable `arg` that was assigned the value `"test"` is out of scope and thus unavailable to any code but the function `inner` that is assigned to `func`. The call `func()` in line 10 will print `"test"` on the console. If the outer function is called a second time, a new inner function with a new closure is created.

3.2 *For-in* loops

For-in loops in JavaScript iterate over properties and not values

For-in loops (also known as *for-each* or *foreach* loops) in most languages that have them allow to iterate over a collection without an explicit counter variable. The common usage is `for (element in collection) {...}`. This is not so in JavaScript. JavaScript instead allows to iterate over properties of arbitrary objects, not just arrays; the syntax is `for (property in object) { ... }` with `variable` getting assigned the *property identifiers* of the object. This incidentally means that, when iterating over an array, the variable does not get assigned the values contained in the array (as one might expect) but instead the *keys* of those elements (which for a proper array would just be consecutive integers starting at 0); the values have to be accessed by `array[key]`.

For-in loops iterate over properties of whole prototype chain

Further, the properties assigned to the loop variable are not only the properties of the object, but those of all objects in its prototype chain (see Crockford [2008]), which

can also be a source of bugs. For this reason every object has a method `hasOwnProperty` that can be used to check whether a property belongs to the object itself or to one of its prototypes.

3.3 Type Coercion

JavaScript has two different operators for testing equality, and another two for testing inequality. The operators `==` and `!=` do *type coercion* before comparing, while the operators `===` and `!==` do not. Similarly, condition tests in *if statements* and *while, do-while* and *for* loops coerce the result of the test to a boolean value if it is not boolean already. Unfortunately, some of the coercions are unexpected or even counter-intuitive. For example, the number 0, an empty string and the *null* object are treated as *false*, while an empty array or any other object is considered *true*. The special number value `NaN` *(not a number)* behaves even stranger; it is not equal to anything, not even to *itself*. (`NaN == NaN` yields *false*.) Crockford [2008] uses for values that are coerced to the value *true* the term *truthy values*; those that give *false* values are called *falsy*. The *falsy* values are the following:

- boolean value `false`
- empty string `''`
- number 0
- special number `NaN`
- `null` object
- special value `undefined`

Chapter 4

Implementation

In this chapter we present the specifics of our implementation. First, we present the rationale for some of the design decisions of our solution. Next we explain how the communication with client applications is organized. And lastly, we describe in more details precisely how the information is generated that is sent to the client.

4.1 Design Rationale

Instrumenting execution environment might lead to lock-in on a custom version

One possibility for designing an application to facilitate live coding would be to instrument the execution environment in which the code is run. If properly executed, this would likely allow the live coding editor to access much of the internal state of the executed code. Any kind of preprocessing of the code would also be unnecessary. Unfortunately, this approach has also its drawbacks. If we had chosen to instrument Node.js and the V8 engine used by it, this would have required significant alterations to the core code of those programs, alterations that might have broken essential functionality (and would have required extensive testing to make sure that they did not do so). It would also mean that the program would only work with a custom version of the runtime environment, that would need to be distributed alongside it. Any changes, bugfixes, and

updates to it would have to be replicated in our version, or one could not develop code for newer versions. This might be quite enough for a prototype or for evaluation purposes, but not for any kind of productive use.

We instrument the code and run it in Node.js

Instead, we decided to *instrument the code* by inserting custom method calls (we refer to them as *callbacks*) into it. To be more precise, we generate a *syntax tree* (AST, abstract syntax tree) out of the code, add the appropriate changes to the AST, and then generate new code out of the AST. This code is then executed in a *sandbox*, and when those callbacks are called, they send back information about the internal state of the executed program. This allows our program to be independent from changes in the runtime environment (as long as those changes do not break backwards compatibility). As a runtime environment we use Node.js[1], the program is written entirely in JavaScript, which also means that it runs on any platform for which there is a Node.js distribution.[2]

4.1.1 Server Architecture

Server communicates via WebSockets with clients and spawns separate runtimes to execute code

We built the application as a server to which a client can connect and send code to, and then receives feedback about the runtime state of the program. The server consists of two major components (see Figure 4.1). First there is the controller component that communicates with the client on one end and spawns child processes on the other, and relays the messages between the both of them. Second is the runtime that instruments the code, sets up the sandbox, and executes the instrumented code in it. The controller communicates to the client via a WebSocket connection. We chose it so the connection from the client has to be established once and messages can be passed between client and server indefinitely, thus significantly reducing overhead compared to an HTTP connection. After receiving new code from a client, the controller creates a new runtime as a *child pro-*

[1] http://nodejs.org

[2] At the moment of writing, there are Node.js binary distributions for Microsoft Windows, Mac OS X, Linux and SunOS, and a source code distribution.

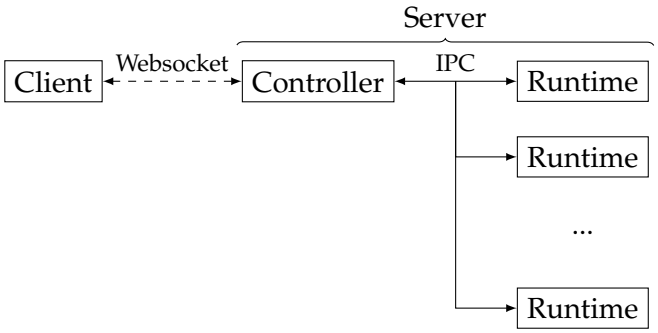

Figure 4.1: Server architecture

cess. This happens through built-in Node.js functionality, the *inter-process communication (IPC)* between the main process and the child processes is provided by the runtime environment.

Executing code in separate processes allows to stop execution

The separation of the actual code execution into child processes was necessary because JavaScript, and accordingly Node.js, has no multithreading support. If the code that is executed in our application contains an infinite loop or hangs for any other reason, then, if it was executed directly in the server process, there would be no way of stopping the execution without shutting down the whole application. The child processes, on the other hand, can be shut down separately. For this, the server accepts messages that allow to shut down a particular runtime, if, for example, the client application believes that the execution will not terminate, or is simply no longer interested in the messages from that particular runtime. If a client closes the socket or shuts down, all runtime processes associated with this client are stopped as well.

4.1.2 Performance Considerations

Performance of instrumentation enhanced by using only one pass over syntax tree

There are several possible performance issues that had to be considered during the application design. First of all, there is the instrumentation of the code. We tried to limit the time spent on instrumentation by using a single pass over the syntax tree. This necessitated the separation in the

code of semantically related parts. Another possible issue is the running of the instrumented code versus the normal execution in Node.js. We evaluate both those issues in section 5.3 "Performance". The last question related to performance is the possible communication overhead between the client and server, and between the server controller and the runtime processes. The possible problems and our solution are presented in the following section.

4.1.3 Third-Party Modules

Applications and third-party modules we used for our application

In our application we made use of the following Node.js modules: *ws* (WebSocket module) for communicating with the client(s), *Contextify*[3] for executing the code in a sandboxed environment, *child_process* (built-in) for spawning child processes in which the instrumentation and execution is performed (see section 4.1.1 for details), *cycle.js*[4] for writing cyclical structures into the JSON format, *Esprima*[5] for parsing the code into the syntax tree, and *Escodegen*[6] for generating code out of the instrumented AST. We used *Contextify* instead of the built-in Node.js module *vm* (which serves the same purpose) because of its better handling of the global object, thus ensuring that most programs will run as if they were simply started as a Node.js program. As a client during development we used the Brackets[7] code editor because of its easy extensibility, with a custom extension that allowed sending code, receiving messages and viewing them in the debugging console.

4.2 Communication

Messages are JavaScript objects converted to JSON

As already mentioned in section 4.1.1 "Server Architecture" (p. 16), the server and the clients communicate via the WebSocket protocol. It allows sending messages between a

[3] https://github.com/brianmcd/contextify
[4] https://github.com/douglascrockford/JSON-js
[5] http://esprima.org
[6] https://github.com/Constellation/escodegen
[7] http://brackets.io

server and a client in full duplex mode. The messages we send consist mostly of JavaScript objects that are converted into a textual representation in the JSON format. All of the objects have a property named `type` to easily determine the content in the message.

4.2.1 Sending Code

The server expects the code in messages of the form

```
{
   type:      "code",
   code:      code,
   [options:  options object,]
   id:        id
}
```

The property `code` should be a string containing the JavaScript code that is to be executed. The property `id` must contain an identifier that will be used to reference all messages associated with this particular code. This id can be either a number or a string and will later be referenced in all messages sent to the client. This way the client can send new code before having received all messages for an older version, and ignore all messages for older versions without processing them. The id cannot contain the *number sign* (#, also called *hash* or *pound* sign), the reason is explained in the next subsection; in our example client we simply use the numeric value of the timestamp when we send the code, converted to a string.

Code and id are mandatory

Id cannot contain the number sign #

The property `options` is optional and permits the client to exhibit finer control over the behavior of the server. The options should be passed as another object with the different settings as boolean properties. If the *options* object or any particular option property is not present, the server will assume default values. In the current version, two options are implemented: `stopOldRuntimes` (default: *true*) and `disposeAfterExit` (default: *true*).

Message can contain options for the server

`stopOldRuntimes` lets the server stop old child processes for this client

The first option, if *true*, tells the server to stop all previously started child processes associated with the client (or rather, with the current socket). If `stopOldRuntimes` is *false*, the child processes will continue to run until the server is stopped, the connection to the client in question is cut, or a child process is stopped manually (how to do that will be explained in subsection 4.2.2).

`disposeAfterExit` lets child process dispose of the context in which the code runs

The option `disposeAfterExit` concerns the sandbox in which the code is executed. If the option is set to *true*, the sandbox is destroyed immediately after the program exits, and the memory can be freed. This is a sensible option for most simpler programs. But an exit from the program occurs after the main code was executed, which does not necessarily mean that the program itself is finished. If the program has deferred some code from execution by using the methods `setTimeout` and `nextTick`, or has some callbacks waiting for events, those will only continue running if the sandbox is *not* destroyed. In such cases the option has to be set to *false*.

4.2.2 Controlling the Server

Message `stopRuntime` can stop a particular child process

Our application accepts other types of messages beside `"code"`. Those can be used to send commands to the server, even during execution of previously sent code. The current version supports only one such message type. The `type` property has to be set to `"stopRuntime"` and the message has to contain a property `runtimeID` with the identifier of a previously started runtime. After receiving the message, the server will attempt to stop the child process with this id. It is useful for stopping the execution of code that has, for example, entered into an infinite loop.

4.2.3 Grouping

Blocking vs. non-blocking message relaying

As already mentioned, JavaScript, and accordingly Node.js, has no multithreading support, thus the execution of code that potentially can contain infinite loops

can be challenging. The two possible solutions that we considered were either to relay any messages from the runtime immediately as they arrive (this is referred to as *blocking*), or to defer the sending of the messages to the next run of the event loop of the execution environment using `setTimeout(function, 0)` or `process.nextTick(function)` (the *non-blocking* way, which is the preferred programming style for Node.js applications).

Sending immediately gives faster feedback, but has large overhead

The first case allows the client, and consequently the user, to receive immediate feedback. But it has the unfortunate downside that, since every seconds hundreds of messages can be generated, the overhead for sending the messages either over the WebSocket connection or the IPC channel between the controller and the runtime, small as they may be for any single message, add up to a quite significant amount and become by far the dominating performance limiting factor.

Deferring messages only gives feedback after program exit

In the second case the actual sending of the messages gets postponed until the next run of the event loop and thus does not significantly slow down the execution of the code (thought the overhead still appears when they are actually being sent). Unfortunately, the next iteration of the event loop is only processed when the execution of the instrumented code terminates. The code sent from the editor may contain infinite loops or other constructs that prevent the code from terminating, and since we cannot reliably detect such cases (the halting problem being unsolvable), the execution has to be stopped in such cases. The only way to do it without stopping the server itself would be to shut down the runtime executing that particular code. If the sending of the messages is postponed, those messages never get sent to the client and thus the user would have absolutely no feedback as to why his code did not execute.

Solution: send messages blocking, but 100 at a time

So we have to send those messages during the execution, but without generating too much overhead. The solution we used was to buffer the messages generated by the runtime and send them off in packages of 100 as soon as enough have accumulated. We chose this number as a compromise between reduced overhead and promptness of

feedback. Smaller packages (for example with 10-20 messages) would still lead to significant overhead, while larger ones would slow down the perceived responsiveness of the server and thus go against the notion of live coding. On faster computers that are able to execute the code faster, it might be possible to raise the size of the package even higher without significantly sacrificing speed of feedback. But a size of hundred has shown to be a good compromise during our testing.

Grouping messages lowers overhead, leading id in message allows to recognize "interesting" message before parsing

These packages are themselves styled as messages, with the type `"message array"` and the messages contained in an array under the property `"messages"`. The client can send new code before having received all the messages from a previous code version, so it has to be able to recognize messages that are interesting to it (e.g., the messages for the last sent code). For this, the message array message contains an *id* field with the identifier that was attached to the code (see subsection 4.2.1). Since JSON parsing is relatively costly, it proved beneficial to be able to discard a message before parsing it, though. To enable this, we decided to prepend the id to the message array message after converting that to JSON; it is separated from the rest of the message by the number sign # and can be split off before parsing. The different types of messages are presented in subsection 4.2.5.

4.2.4 Receiving Messages

Error messages are sent out of order

In addition to the normal messages that inform the client of changes of state of the executed application and are contained in the message array, there are also several types of feedback and error messages. (By those we do not mean errors generated by the executed code, but those happening in our application itself, like problems with the instrumentation or execution of the code.) They are sent separately from the message array as soon as they happen. The different kind of errors are described in subsection 4.2.6.

4.2.5 Information Messages

Console replacement allows text output

Writing text to the console via `console.log()` and `console.error()` is a popular form of providing feedback or debugging in Node.js. Simply providing the normal console object to the code executed in the sandbox would output those messages to the console of the server, which would not be very useful to client applications and consequently to the developers using them. To allow those messages to be displayed by the client as it sees fit, we chose to replace the console that we pass to the sandbox object. Our console replacement supports the methods `log()` to write information messages to the console and `error()` to write errors to the error console (often they are identical, but web browsers, for example, tend to style error messages differently). Other methods are currently not implemented (see also section 5.2 "Known Issues and Limitations" (p. 42)), e.g. console *input* is not possible.

All message types in a message array have type and location information

After unpacking the message array the client can encounter several types of messages. All of them have a property `type` (the message type) and a property `loc` (the location of the part of code that caused the message in the *original* code), with the exception of `log` and `error` that do not have a location property. The location message is structured the following way:

```
{
   start:  { line, column },
   end:    { line, column }
}
```

It logs the start (line and column) and end point (line and column) of the element in question. An overview of all messages generated by the instrumented code is presented in Table 4.1.

Type property	Properties	Description
program finished	loc	Program execution was finished
variables init	vars [name, value], loc	Variables vars[].name initialized with values vars[].value
assignment	name, value, loc	Variable name was assigned value
update	name, value, loc	Variable name was updated to value
function enter	name, argNames [name], argValues [value], loc	Function name was called with parameters argNames, argValues
function exit	loc	Function exited without return value
return	value, loc	Function exited returning value
if-test	result, value, loc	Test of if-condition evaluated to value, boolean result
switch-condition	value, loc	Switch condition evaluated to value
while-loop init	loc	While loop started
while-loop-test	result, value, loc	Test of while loop evaluated to value, boolean result
while-loop enter	loc	While loop iteration started
while-loop exit	totalLoopCount, loc	While loop exited after totalLoopCount iterations
for-loop init	loc	For loop started
for-loop-test	result, value, loc	Test of for loop evaluated to value, boolean result
for-loop enter	loc	For loop iteration started
for-loop exit	totalLoopCount, loc	For loop exited after totalLoopCount iterations
for-in-loop init	loc	For-in loop started
for-in-loop enter	loopVar, value, loc	For-in loop iteration started, loop variable loopVar has value
for-in-loop exit	totalLoopCount, loc	For-in loop exited after totalLoopCount iterations
log	text []	Writing text to console
error	text []	Writing text to error console
begin try-catch	loc	Begin of a try block with a catch part
begin try-finally	loc	Begin of a try block without a catch part
throw	loc	Exception was thrown
catch	exception, loc	Exception was caught in a catch part
end try	loc	Try block exited (no exception or exception handled)
uncaught exception	exception, loc	Exception was not handled

Table 4.1: Overview of all messages generated by the instrumented code

4.2.6 Feedback and Error Messages

Runtime informs after start, after successful or unsuccessful instrumentation and execution

To inform the client application(s) of the current state of execution, the runtime sends the client the following messages. A first message is sent by the runtime process (see also section 4.1.1 “Server Architecture” (p. 16)) immediately after it has started. After the instrumentation has been completed successfully, a log message is sent with the text `"Code successfully instrumented in t ms."`; *t* is the time in milliseconds that the instrumentation has taken. An error during the instrumentation leads to the message `"Error instrumenting code: exception"`; *exception* is the exception object that lead to the error. Similar messages are generated for the execution of the instrumented code. If an error happens during the execution and some messages from subsection 4.2.5 “Information Messages” have accumulated but not yet sent, they are sent to the client *before* the error message, to allow the client as much feedback as possible and also possibly locate the source of the error more precisely. The messages are styled in similar manner as the `log` and `error` messages mentioned in subsection 4.2.5, with the addition of the *id* of the code (see also subsection 4.2.1 “Sending Code”).

4.3 Instrumentation

We parse the code, insert callbacks into the AST and generate new code from it

The instrumentation is done by parsing the code, going recursively through the AST, inserting calls to a specific function at appropriate places in the tree, and afterward generating new code back from it. The parsing is done with the Esprima parser, for the code generation we use Escodegen. Additionally, after generating the instrumented code from the AST, we wrap all the code in a *try-catch block* to capture any unhandled exception and inform the client about it. After the block we insert an additional callback call to signal that the execution of the code has finished. Esprima is configured such that it adds location information to every element in the tree. This information is retained during the instrumentation and added to the messages sent to the client. The client can then use this information to find

the position in the original code that corresponds to the received message.

The following elements are instrumented in the code:

- Variable value changes
- Variable declarations
- If and switch statements
- Functions
- For loops
- For-in loops
- While loops
- Do-while loops
- Try-catch blocks

Replace single statements with blocks in control structures

In all the instrumented constructs whenever the instrumentation code encounters a situation where a non-block statement is used in the code where a block statement was possible, such as in `if (a < b) doSomething();` instead of `if (a < b) { doSomething(); }`, we replace the non-block statement with a block statement and insert the original statement into the block (unless the superseding construct is a block itself). This does not change the semantics of the code and enables us to insert instrumentation code into the block.

Following are the details of the different instrumentations. In all examples shown in this section the location information is omitted from all callback function calls for brevity. Furthermore, some variable names and function calls to our instrumentation code are shortened for better clarity.

4.3.1 Variable Assignments and Updates

Assignments and update operations change variable value

There are two kinds of simple operations that change variable values in JavaScript: *assignments*, that overwrite a variable value and are denoted by an equals sign (=), and *updates*, that change a variable's value relative to its original value. The latter ones are the simple *increment* and *decrement* operations (++ and –), and the combination operations +=, -=, *=, /= and %= that combine an operator with an assignment. They all cause a change in the value of a variable on the left side of it, and so the instrumentations are essentially equal, aside from the type of the message sent to the client (*assignment* and *update* accordingly). Besides the type the message contains the name of the variable and the new value that it has after the operation.

Assignment or update can be a separate statement or part of a sequence

Assignments and updates can be found as separate statements, mostly found in block statements, or as part of a sequence expression (several expressions separated by commas). For statements found in blocks we simply insert a callback immediately after the assignment/update. For sequences we insert a callback into the sequence after the operation in question. The latter is, for example, the case in *for-loops*, where an increment operation is often found in the afterthought part (see also subsection 4.3.5).

Assignments in *if statements* are handled elsewhere

An assignment can also occur in the condition of an *if-else expression*. Most often this constitutes an error by the programmer. Indeed it is a quite common and, more importantly, hard to spot mistake. An assignment gets inserted instead of a comparison, such as writing `if (a = 1) ...` instead of `if (a == 1) ...` or `if (a === 1) ...`, which is still valid JavaScript code as far as the runtime environment is concerned, but is often not what the programmer intended. In this case we do not provide an *assignment* message, but this is handled with the message for the *if statement* (see subsection 4.3.4).

4.3.2 Variable Declarations

Declarations create variables and can initialize them

A *variable declaration* in JavaScript is a statement that creates new variables in the current scope. Every declaration statement can contain more than one declaration (for example, `var a, b;` declares two variables at once. Every declaration can also *initialize* a variable, (e.g., `var a = 10, b = [];`), which assigns a value to the variable. If the variable is not initialized, its value after the declaration is the special value `undefined`. As explained in subsection 3.1.1 "Function Scope and Hoisting" (p. 10), the *hoisting* mechanism semantically moves the declaration to the top of the scope (without moving the assignment).

We insert callback after the declaration with a list of all declared variables and their values

The instrumentation of a declaration statement is very similar to an assignment or update (see previous subsection). We insert the call of the callback function after the declaration; the type of the message is *variables init*. The main difference is that we possibly have to send more than one variable and value. Thus we group the declared variables and their initial values (if they have any) into objects of the form { `name:` *`name`*`, value:` *`value`* }. Those objects are then collected in an array, which is attached to the sent message under the property `vars`.

4.3.3 Functions

Functions provide feedback on entry and exit

In case of functions we want to provide feedback as to when a function or method has been entered and left. Additionally, we determine which parameters have been passed to it during the call, since in a language like JavaScript this can be a source for errors: the parameter list is not part of the function signature and is not checked during compile time nor during run time. A function can even work with more parameters than are in the parameter list. To enable this, every function has an implicit variable named *arguments* in its scope that contains all parameters with which the function was called in a pseudo-array. We use this variable to easily access all parameters and add them to the message sent to the client application. In addition to that, we gather the list of the declared parameters of the function

(names only) and add them too. All this is sent in a *function enter* message that is inserted as the first statement in the function.

PSEUDO-ARRAY:
A *pseudo-array* is an object that looks like a normal array: it has elements listed under properties with sequentially numbered integers as names, starting at 0, and a non-enumerable property named *length*. However it is not an instance of the *Array* class and lacks all methods of an array.

Definition: *Pseudo-array*

As to the function exits, there are several ways to leave a function:

Functions Without *Return* Statements

One way is to simply finish executing all statements in the function; this is the case for functions that do not return any value, sometimes called *procedures*. We recognize functions that do not have a *return* statement as their last statement as such and add a call to our callback function as the new last statement that sends a *function exit* message. Of course it can happen that a function contains *return* statements embedded in *if-else* conditional statements or other control structures such that all possible execution paths will inevitably call one of them. In that case the instrumentation algorithm will still (erroneously) recognize the function as a procedure and embed a callback at the end. But since, as previously stated, all execution paths lead to one of the *returns*, this last callback is never actually called during execution.

Function without return value exit with *function exit* message

Return Statements

Another way to exit a function is to execute a *return* statement. Those can, but do not need to have a return value. Without a return value the function essentially becomes a

Return values are assigned to temporary variable

procedure, thus we simply insert a *function exit* message before the return. If the *return* has a value, we send a *return* message and pass this value along with it. Of course, this value can be just a literal (like a simple number, string or an in-place object or array) or an identifier (variable or function name), in which case embedding it into the message becomes quite simple. But the return value in JavaScript can also be quite complicated, with logical constructs (like the ternary conditional operator *?:*), function calls and even function definitions. In such a case we have to be careful to avoid any possible side-effects that can alter the behavior of the instrumented program. To achieve this, we create a temporary variable, assign to it the original return value (thus any possible calculations and function calls get executed at that point in the instrumented program), add a callback function call with the type *return* and the value of the temporary variable, and replace the original value of the return statement with the temporary variable. As an example,

```
1 function func() {
2     ...
3     return func2() + 7;
4 }
```

becomes

```
1 function func() {
2     ...
3     var temp = func2() + 7;
4     callback({ type: "return", value: temp });
5     return temp;
6 }
```

Returns may lead to exits from loops or other control structures

Independently of whether the *return* statement has a return value or not, it can be nested deep in the control structures of a function and, for example, if the *return* is inside a loop, the function exit also represents an exit from the loop. How such cases are handled is covered in subsection 4.3.6. One has also to keep in mind that a return does not necessarily mean that no more code in the function is executed. If the *return* statement is in a *try[-catch]-finally* block, all the code in the *finally* block is executed before exiting the function, and the *finally* block itself can contain *return* statements and other control structures.

Exceptions

Another thing that can (but not necessarily must) lead to leaving a function is an *exception*. This can be an exception in the code of the function itself (e.g., in case of an attempt to call a non-existent method of an object), an uncaught exception in code called from the function, or an explicit *throw* statement (in case of an error condition arising, for example). For finding such cases, *try-catch-finally* blocks are also instrumented. Since exceptions can also lead to the termination of loops, the precise handling of such cases is covered in subsection 4.3.6.

Exceptions can also lead to leaving a function

4.3.4 *If* and *Switch* Conditionals

If statements allow branching of program flow depending on a binary condition. In JavaScript, the condition can be not only a boolean literal (*true* or *false*) or an operation that yields a boolean value (e.g. `a < 5`), but any value that can be *coerced* to a boolean value, which is essentially any value in JavaScript. Some values are converted to the boolean *false* value (so called *falsy* values, those are 0, *NaN*, empty string, *null* and *undefined*), the rest to *true* (*truthy* values; terminology by Crockford [2008]). The value of an assignment is the new value of the left hand side variable, which can in turn be brought to a boolean value. Thus in `if (a = 5) ...` the branch of the *if* statement will be executed, but in `if (a = 0) ...` it wont. This can be used purposefully by the programmer, for example, to test for the existence of an object, but can as easily constitute an error.

If statements

A *switch statement* is used for a similar purpose, only it allows more than two branches at the same time and the branching condition is not confined to a boolean value. The condition, that in JavaScript can be more than just an identifier, is evaluated first, and then the appropriate branch is executed. JavaScript allows to use any type of variable as a possible case, not just ordinal types like integers or single characters. The value of the switch condition is also valuable information for the developer.

Switch statements

Instrumentation for ifs and switches wraps condition in anonymous function

The way we instrument the test of the *if* statement and the condition of the *switch* is essentially identical, apart from the type of the message that is sent. We send the test result without changing the program semantic by wrapping the condition in an anonymous function, with the condition as a parameter, that is executed immediately. This function calls the callback function and returns the value of the condition, both the original one and the boolean value. The following example demonstrates it:

```
1 if (a < 5) {
2   ...
3 }
```

is transformed into

```
1 if ((function (result) {
2     callback({ type: "if-test", result: (result ?
          true : false), value: result });
3     return result; }) (a < 5)) {
4   ...
5 }
```

Returning original and boolean values might help to find mistakes from wrong assumptions about *true* and *false* values

This allows the programmer to see the result of the test condition *before* it was coerced to a boolean value (if it was not a boolean already). An assignment, for example, has as its value the value that was assigned to the variable and will appear as such in the message. Returning both the original and the boolean value also allows to spot errors resulting from misunderstanding the concept of *truthy* and *falsy* values in JavaScript. For example, the empty string and the number 0 are falsy and thus will be converted to *false*, but the empty array `[]` and the empty object {} are truthy. This is part of the JavaScript standard, but might seem counter-intuitive to some programmers, and providing those values in live coding might help to understand why an application for example enters a specific branch.

4.3.5 Loops

JavaScript has different types of loops

As any modern programming language, JavaScript has several types of loop constructs. They are commonly referred to as *while loops*, *do-while loops*, *for loops* and *for-in*

loops. They are often used to iterate over arrays or to execute some operation until some condition is met, and sometimes are even interchangeable, though the *for-in* loop has some unusual and unexpected properties (more on it in subsection 4.3.5) that set it apart from the other loops and seemingly similar constructs in other languages.

While and *Do-while* Loops

Instrumentation of *while* and *do-while* loops

While loops and *do-while* loops in JavaScript are quite similar; they both repeatedly execute the loop body as long as some condition is met, the only difference being the point in execution when this condition is evaluated. *While* loops check the condition at the beginning of every iteration (thus it is possible that the loop never gets executed at all), while *do-while* loops test it at the end (thus guaranteeing that the loop gets executed at least once). The instrumentation of the loops is mechanically identical: a callback is inserted immediately before the loop to signal the loop start (needed for termination handling, see subsection 4.3.6), another callback is inserted in the beginning of the loop for the loop entry and a last one just after the loop for the loop exit. The following example shows this for the *while* loop:

```
1 while (condition) {
2   ...
3 }
```

becomes

```
1 callback({ type: "while-loop init" });
2 while (instrumented condition) {
3   callback({ type: "while-loop enter" });
4   ...
5 }
6 callback({ type: "while-loop exit" });
```

Instrumentation of the test same as for *if statements*

The same considerations apply for the loop condition of the *while* and *do-while* loops as for the test of the *if* statement, so the instrumentation, that is only hinted to in the example is done by using the same method (by inserting an anonymous function around it, see subsection 4.3.4).

For Loops

Syntax and usage of *for loops*

A *for-loop* is a loop construct consisting of an *initialization* statement (executed once at start, commonly for initializing a loop counter variable), a *condition* expression (evaluated before every iteration and converted to boolean value, if value is *falsy*, loop is exited) and an *afterthought* statement (executed after every iteration, usually for incrementing or decrementing the loop variable). Ofter, *for loops* are used like this: `for (i = 0; i < max; i++) { ... }` However, the language puts virtually no limits as to what can be put into the respective parts. For example, the afterthought can update more than one variable, or the initialization can not just initialize the variable, but also declare it. The latter is quite common for languages with a C-style syntax but is discouraged in JavaScript. The reasons for this are *function scope* and *hoisting* (see subsection 3.1.1) that might lead to hard to spot errors.

We extract variable declaration out of loop initialization

The declaration of a variable in the initialization part of the *for-loop* gets also hoisted to the beginning of the function. This behavior can lead to errors in the program, so it has to preserved by our instrumentation. We would want to insert callbacks into the initialization of the loop (in case of variable assignments or function declarations, for example), but appending them to a variable declaration would be syntactically invalid, i.e. turning `for (var i = 0; i< max; i++) { ... }` into `for(var i = 0, callback(...); i < max; i++) { ... }` is not possible. To solve this we decided to pull an eventual declaration statement out of the *for-loop* and insert it immediately before the loop (this declaration also gets hoisted up the function, so the semantics remain unaltered), while turning a possible initialization into an assignment and keeping it in the loop initialization. Syntactically the initialization statement gets turned from a declaration into a sequence expression, so that we can append additional expressions to it. Similarly, the afterthought part is also converted into a sequence expression, if it is not one already.

Instrumentation of test and body

As for the test, here the same considerations apply as for the *if statement* and the *while loop*, thus it is instrumented in the same manner (for details see subsection 4.3.4). We then recursively instrument the initialization and the afterthought parts and the body of the loop (this is where all callbacks for variable assignments and updates get inserted), insert a *for-loop init* callback as the first element in the initialization sequence, a *for-loop enter* callback as first element in the loop itself and a *for-loop exit* callback immediately after the loop. Example:

```
1 for (var i = 0; i < max; i++) {
2     ...
3 }
```

becomes

```
1 var i;
2 for (callback({ type: "for-loop init" }), i = 0,
      callback({ type: "assignment", name: "i", value:
       0 }); i < max; i++, callback({ type: "update",
      name: "i", value: i })) {
3   callback({ type: "for-loop enter" });
4   ...
5 }
6 callback({ type: "for-loop exit" });
```

For-in Loops

Instrumentation of *for-in loops*

As we previously explained in section 3.2, *for-in* loops in JavaScript iterate not over elements of a collection or property values of an objects, but over property *identifiers*. Despite this, the instrumentation of the `for-in` loops is quite simple and very similar to the `while` loops (see 4.3.5. Callbacks are inserted before the loop (initialization), as first expression in the loop (loop entry) and immediately after the loop (loop exit). The only difference is that, since we know the loop variable, we can insert it into the message, as in the following example:

```
1 for (key in array) {
2   ...
3 }
```

becomes

```
1 callback({ type: "for-in-loop init" });
2 for (key in array) {
3   callback({ type: "for-in-loop enter", loopVar: "
        key", value: key });
4   ...
5 }
6 callback({ type: "for-in-loop exit" });
```

4.3.6 Loop Terminations and Multiple Function Exits

Problem Description

Several ways to leave a function

In JavaScript, as in many other modern programming languages, there are several ways to exit a function, voluntarily or involuntarily. In JavaScript, it is possible to terminate a function at any time through the use of the *return* command, which can have an optional return value. This will of course also exit any loops in that function that the program is currently in.

Loop control commands

Similarly, there exist several commands to control the execution of loops. Those are *label*, *continue* and *break*. *Continue* stops the execution of the current iteration and goes immediately to the next one, which of course might not happen if the loop condition is no longer met. The effects of this command are already covered by the instrumentations introduced previously in section 4.3.5 “Loops”. The *break* command on the other hand completely stops the execution of the loop it is in. (The *break* command is also used in *switch* statements, but that case is completely irrelevant to our instrumentation.) The *label* command allows to assign a label to a loop; this label, in conjunction with *break*, allows to terminate several loops at once, up to the labeled one.

Exceptions can lead to leaving loops and/or functions

The situation gets even more complicated when one considers exceptions. Those can be explicitly triggered by the programmer through the use of the *throw* command or triggered by some error condition during the execution. One way or the other, an exception stops the execution of the current function.

Exception can be handled in a *try* block

If the exception occurred in a *try block*, then what happens further depends on whether the *try* construct has a *catch* part and/or a *finally* part. If there is a *catch* part, the execution jumps to it, where the exception is handled; if there is also a *finally* part, it will be executed after the *catch* block, even if the exception was rethrown. (The *finally* part is guaranteed to be executed under almost any circumstances.) If the exception was not rethrown, the exception is considered handled and execution continues after the *try-catch* block; only the loops the program was in that are in the *try* block are exited. Without a *catch* part, the *finally* part (that has to be there in this case) gets executed, and then, since the exception is considered unhandled, the function is terminated and the exception is rethrown.

Unhandled exceptions terminate all loops and functions up to the place where they are caught

If an exception is unhandled, be it because it was not caught in the function where it occurred, or it was rethrown further up the execution stack, it will lead to leaving all currently executed loops in the function, the function itself and all the executed loops and functions up to the place where the exception is caught and handled. If the exception is not handled anywhere, the whole program will terminate. Since we decided to inform the client about the exiting of functions and loops, we have to somehow store the information of currently executed loops and functions, and be able to send the appropriate exit messages to the client in case of an exception or *return* statement.

Solution

Stack in closure of callback function tracks functions, loops and try-catch blocks

Since a program execution in JavaScript consists of nested function calls and structural constructs (e.g. loops and *try-catch* blocks), using a *stack* to track them is an obvious solution. This stack is implemented as a JavaScript array and kept in the *closure* (see subsection 3.1.2 “Closure” (p. 11)) of our callback function. It it thus not in scope and therefore invisible to any code that calls the callback function (so it cannot clash with any code that we instrument and execute), but only visible to the callback function itself. We use the stack to track all enters and exits of *functions*, *loops* and *try-catch blocks*.

Enters push new items on stack, exits pop them

We put a new item on the stack on any enter into a function, a *try-catch* block (*not* a *try-finally*, *try-catch-finally* is handled as a *try-catch*) or a loop (on *loop init* message). The stack items store the type and location information from the callback function, for the loops we also add a property `count` to track the number of iterations. This number is increased with every new iteration (on *loop enter* message) and sent to the client with the *loop exit* message. (It is purely a matter of convenience since the client can just as easily track that information for itself.) Any regular exit from a loop will generate a *loop exit* message that will take the item from the stack and append the loop count to the message sent to the client. It does not matter whether the loop was left by no longer passing the test or through the *break* command, since the callback call that sends the *loop exit* message is inserted *after* the loop and thus gets executed in each case. (The difference will still be visible in the list of messages received by the client: an exit through failing the test will have a *loop test* message before the *loop exit* message, unless it is a *for-in loop*.)

Function exits also exit any active loop and *try-catch* block

For functions it gets slightly more complicated, since a function exit can be in form of a *return* statement. (And those *returns* can generate a *return* or a *function exit* message, depending on whether they return a value or not.) A *return* can easily be found inside a loop or several nested loops, and/or in *try-catch* blocks. So when a *return/function exit* message arrives, the top stack item might not be a function. Those other items have to be all taken down from the stack and the appropriate messages have to be generated *before* sending the *function exit* message to the client. If the *return* is in a *try[-catch]-finally* block, the *finally* part will be executed before exiting the function, and there can be additional *returns*, that, when executed, will break our stack, since two *return* messages will arrive for one function. Unfortunately, we have not found a solution to this problem. We have to rely on this not occurring often, since the *finally* construct is meant for freeing critical resources and not for complicated flow control structures.

Exceptions terminate loops and functions until they are handled

Exceptions are the most complicated case for our solution. An exception, whether explicitly thrown, coming from other executed code, or from the runtime environ-

ment, will exit any loop and function on the stack up until it is caught by a *try-catch* construct (and not rethrown). The only code that gets executed between the generation of an exception and the catching of it are possible *finally* blocks on the way. To handle this, we put items on the stack for every *try-catch* block, but not for *try-finally* (*try-catch-finally* is handled like a *try-catch*). A *throw* message will pop all items from the stack until it encounters a *try-catch* item (that means that the exception is handled there) or empties the stack (which means that the exception was not handled). For every popped item an appropriate message is generated and put into the message queue. In case the exception is handled we leave the *try-catch* item on the stack; it will be removed by the *catch* message. The *catch* message does essentially the same as *throw* (this is needed for exceptions that were not generated by a throw), but does remove the *try-catch* item. The last possible message is *end try-catch*, it signaled by a callback inserted after a *try* block. After this message, the runtime looks at the topmost stack item and takes it down if it is a *try-catch*.

Chapter 5

Evaluation

In this chapter we want to evaluate the capabilities and performance of our application with respect to its applicability for user studies of different visualization solutions, and for practical use. Since our was not to create and evaluate a particular visualization, but to instead provide a robust technical base for such studies, we can only present a descriptive overview of the qualitative capabilities of our program. We can however speak of its limitations and possible solutions for them. Lastly, we consider the issue of the *performance* of our solution. Since *live coding* places high demands on the swiftness of feedback to the user (it is indeed implied in the name itself), it was important to us to measure the speed of the different parts of our application.

5.1 Capabilities

We tested the capabilities of the solution with several types of files

To test the capabilities of our live coding server, we used several types of programs. Firstly, we used synthetic examples to test particular issues or instrumentation types, e.g., the behavior in case of exceptions that is described in subsections 4.3.3 “Exceptions” (p. 31) and 4.3.6 “Loop Terminations and Multiple Function Exits” (p. 36). Secondly, we used smaller programs that are representative of real life applications for live coding. One such example is the

merge sort algorithm that is in its completeness presented in appendix A "Testing code" (p. 51). Even this fairly simple algorithm, sorting an array of only 7 elements, generates 164 different messages, which goes to highlight the level of detail that the server is capable of providing. A slightly modified version of this code was used for the performance tests in subsection 5.3.2 "Execution". Thirdly, we used real Node.js applications and modules to test the correctness and completion of our instrumentation, and as a way to measure the performance of the instrumentation (further described in subsection 5.3.1).

5.2 Known Issues and Limitations

No console input

Our application has some limitations as to what types of code it can execute and provide feedback on. One of the issues is interactivity: we can only *output* data to console, capturing console *input* is not yet possible. The reason for is the *console replacement* we presented in subsection 4.2.5. To solve this issue, the console replacement would have to be enhanced with the capability of sending data to the application, while also provide means to send those input data from the client application to the console replacement.

Only Node.js programs

Another limiting factor is the fact that we run the code in a Node.js environment, consequently we can only execute programs that are compatible with it. This issue cannot be solved easily. To simulate a different environment, a *web-browser* for instance, the sandbox that we use to execute the code in would have to be furnished with objects and methods that are part of such an environment. (For a web-browser environment, this would be, among others, the `window` object with all its methods.) This replacement objects and methods should also be able to send appropriate information messages to the client application (similar to our console replacement).

Functions cannot be encoded in JSON

The JSON format is not capable of encoding function objects. Since we use JSON to be able to send JavaScript objects over websockets, all functions and their associated closures are lost in the process. It might be possible to send the

source code of the function, but even then the closure of the function would be lost. Unfortunately, we do not believe that a perfect solution can be found in the context of our application architecture.

Breaks in conjunction with labeled loops break the message stack

The *label* command that allows the *break* command to break out of several loops at once, previously mentioned in subsection 4.3.6, is not completely supported. It will not cause problems during the instrumentation, but, if used in the program for breaking out of more than one loop, might break the stack used for sending messages for terminated loops and functions described in subsection 4.3.6. The messages generated by the stack *after* a labeled *break* statement can no longer be relied upon, since stack items from all loops except the innermost will remain on the stack until the next function exit.

5.3 Performance

Testing parameters

All performance measurements were made on a Lenovo T430s laptop with an Intel i5-3320M CPU and 8GB of RAM, running under a 64-bit Windows 7 OS. For time measurements we used the built-in JavaScript timing capabilities that provide millisecond resolutions.

5.3.1 Instrumentation

Instrumentation performance measured on large files

We measured the performance of the instrumentation on 5 files of different sizes. Since for smaller files the instrumentation time varies too much in relation to the timer resolution, we deliberately selected larger files, with sizes ranging from about 850 to almost 19,000 *source lines of code* (SLOC, lines of code without comments and empty lines). We timed the instrumentation for each file 10 times and calculated the mean of the durations. From this, we calculated the ratio of the mean to the number of SLOC to account for the different file sizes.

Table and chart show instrumentation times for different files

Table 5.1 shows the data for files *esprima.js* (from Esprima[1]),

escodegen.js (from Escodegen[2]), *underscore.js* (from Underscore.js[3]), *jslint.js* (from JSLint[4]) and *test.js* (testing file, also from Esprima). The chart in figure 5.1 shows the ratio of *instrumentation time* versus number of SLOC for those files. From the data it is apparent that, though the times generally get larger with larger files, some other factors also play a role. The file *test.js*, for example, though quite large at almost 19,000 SLOC, has a comparatively small instrumentation time of approximately 2.1 seconds. The reason for this may the relatively simple structure of the file, since it consists almost completely of object definitions and has not a single function declaration. Generally we see that the instrumentation time is under one millisecond per source line of code.

Since the time between the client sending the code to the server and the server sending the first feedback messages is largely dominated by the instrumentation, this time has to be small. If we assume that feedback time under $\frac{1}{2}$ second is sufficiently fast, this means that our solution can work with files of at least 500 lines of code, and depending on the structure of the file, 1000 lines might also be fast enough for liveness.

File	SLOC	Mean time [ms]	Mean/SLOC [ms]
underscore.js	848	713	0.841
escodegen.js	1914	831	0.434
jslint.js	2958	2222	0.751
esprima.js	3096	1314	0.424
test.js	18995	2110	0.111

Table 5.1: Performance of the instrumentation algorithm for files of different size

5.3.2 Execution

Performance of execution was measured on three different algorithms

For the evaluation of the execution we decided to com-

[1]http://esprima.org
[2]https://github.com/Constellation/escodegen
[3]http://underscorejs.org/
[4]http://www.jslint.com/

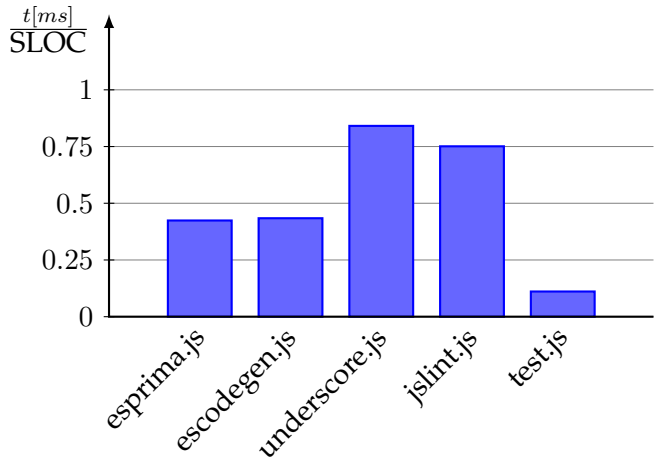

Figure 5.1: Ratio of instrumentation time to SLOC for files of different size

pare the execution time of the instrumented code with the time for the original code under the same circumstances. Since the sorting algorithm shown in appendix A is simply too fast to yield significant results for the sorting of 7 elements, we increased the number of elements to 4000. We measured the running time 10 times for each array size, for original and instrumented code respectively. In addition to *merge sort* we also used two other algorithms: *bubble sort* sorting 1000 elements (the algorithm is slower and generates a lot more messages) and the *babylonian method* for iteratively calculating the square root with 100 000 iterations. The execution times for all algorithms is shown in Table 5.2.

Algorithm	**Original time [ms]**	**Time during live coding [ms]**
Merge sort	25	3 552.7
Bubble sort	16	50 841
Square root	15	5 484.3

Table 5.2: Performance of the execution for different algorithms

Big amount of messages slows down execution

One can easily see that the slowdown is quite significant for all algorithms. Especially bubble sort with over 50 seconds versus 16 milliseconds without our application. The reason for this is the huge amount of messages generated by this algorithm, in contrast with actual computation being performed in the square root algorithm. All messages are processed in the server to preserve their state at the moment of execution, and this processing adds up to quite a big amount of time. We are convinced that through further optimizations this slowdown can be reduced, but it will at least be one or two degrees of magnitude over the original code. This means that during user studies it is advisable to use algorithms that generate less messages during execution (i.e. they perform more taxing calculations instead) and/or use smaller sized input data. We found during development that using 7–10 elements for a sorting algorithm is enough to show whether it works properly, and the slowdown at this size is imperceptible.

Chapter 6

Summary and Future Work

6.1 Summary and Contributions

After studying the available research on the topic and evaluating several possible approaches, we implemented a server in Node.js that accepts WebSocket connections from client applications. After establishing a connection to a client, it accepts messages with JavaScript code, instruments and executes it, and sends the changes in program state back to the client. In addition to the server we implemented a low fidelity client as a plugin for the Brackets code editor for testing purposes; it is only capable of sending JavaScript code currently visible in the editing window and writing the received messages to the debugging console in Brackets. We then evaluated the capabilities of the application and highlighted some limitations that need to be addressed in the future. We also analyzed the performance of the server, showing that there is a considerable slowdown in execution time for programs that generate a lot of messages, mostly those with larger amounts of input data. This has to ba taken into account when designing possible user studies with our application as a backend.

Since the client plugin was only for testing purposes and not applicable for a user study, we cannot provide any concrete data to the utility of our application. Any evidence we have is anecdotal: the application is, at the moment of writing, used as a backend for a user study for a live coding application. This, and possible other future studies will provide more feedback and allow to see possible weaknesses of the current solution.

The main contributions of our work are:

- determining what feedback a user might be interested in during live coding
- designing and implementing a server capable of executing the provided code and returning the information to the clients
- evaluating the technical limitations of our solution and proposing possible solutions

6.2 Future Work

Visualization solutions

The presented solution provides a large amount of feedback and puts fairly little limitations on the executed code. But what possible data might also be beneficial for a developer and how it is presented can only be evaluated in future qualitative and quantitative user studies. Our application provides a stable basis for the evaluation of different visualization in a largely unexplored visualization space.

Console input

There are also possible improvements to the backend itself. Providing means for writing interactive console applications would greatly enhance possibilities for user studies and allow the user to provide input data to an algorithm through an interactive interface instead of writing it into the source code directly. But, apart from the technical limitations (see section 5.2), The precise interaction between the interactive console and the live coding interface have to be evaluated.

Providing feedback for function declarations

JavaScript is a functional language, and so it puts a lot of emphasis on functions. Our implementation provides feedback when a function is called (and the parameters of the call). It might also be beneficial to give feedback for *function declarations*, since the same function in the *code* can represent different *function objects* at runtime. Unfortunately, it has proven difficult to instrument all function declarations consistently, since JavaScript allows to declare a function at many different positions in the code. Such attempts have also lead to significant slowdown of the instrumentation. Therefore we decided to not include it on our application, though it might be beneficial to add such capabilities in the future.

Instrumenting runtime environment

Another possible direction for future work is the instrumentation of the execution environment (in our case Node.js). Something similar was already done by Edwards [2004], but we propose to do this while retaining our server-client architecture, thus not limiting the editor and visualization decisions. Despite its possible drawbacks (see section 4.1 “Design Rationale” (p. 15)), a backend built on top of an instrumented runtime environment might provide much more detailed feedback because of the better over the execution environment. It also might alleviate the performance issues we highlighted in section 5.3 “Performance” (p. 43).

Executing functions separately with provided input

Yet another possibility for future work would be the capability to execute not the whole code, but just functions, while providing them with input data. This input data may be entered by the user, or even gathered from a previous execution of the whole program. This might further accelerate the execution and provide the developer with feedback to those parts of the program he or she is interested in. Unfortunately, the concept of *closure* in JavaScript (see subsection 3.1.2 “Closure” (p. 11)) might severely limit the practicality of such a solution, because the closure of a function would have to be treated as another set of input data.

Appendix A

Testing code

A.1 Babylonian method for calculating the square root

```
1  /*jslint node: true */
2  "use strict";
3
4  function root(number, iterations) {
5      var i, root = 10;
6
7      for (i = 0; i < iterations; i++) {
8          root = (root + (number/root)) / 2;
9      }
10
11     return root;
12 }
13
14 var result = root(2, 100000);
15 console.log(result);
```

A.2 Merge sort

```
1  /*jslint node: true */
2  "use strict";
3
4  var testarray = [6, 3, 8, 0, 4, 7, 5],
5
6  function merge(array1, array2) {
7      var i = 0, j = 0, result = [];
8
9      while (result.length < array1.length + array2.
           length) {
10         if (i <= array1.length && (j >= array2.
               length || array1[i] < array2[j])) {
11             result.push(array1[i]);
12             i += 1;
13         } else {
14             result.push(array2[j]);
15             j += 1;
16         }
17     }
18     return result;
19 }
20
21 function sort(array) {
22     var halfpoint, result;
23     if (array.length === 1) { return array; }
24
25     halfpoint = array.length / 2;
26     result = merge(sort(array.slice(0, halfpoint)),
           sort(array.slice(halfpoint, array.length)));
27     return result;
28 }
29
30 var result = sort(testarray);
31 console.log(result);
```

A.3 Optimized bubble sort

```
1  /*jslint node: true */
2  "use strict";
3
4  var testarray = [6, 3, 8, 0, 4, 7, 5],
5
6  function sort(array) {
7      // Add swap method to the array
8      array.swap = function (i1, i2) {
9          var tmp = this[i1];
10         this[i1] = this[i2];
11         this[i2] = tmp;
12     };
13
14     var n = array.length, newN, i;
15
16     do {
17         newN = 0;
18         for (i = 1; i < n; i++) {
19             if (array[i - 1] > array[i]) {
20                 array.swap(i - 1, i);
21                 newN = i;
22             }
23         }
24         n = newN;
25     } while (n !== 0);
26
27     delete array.swap;
28     return array;
29 }
30
31  for (var i = 0; i < 0; i++) {
32    testarray2 = testarray2.concat(testarray2);
33 }
34
35 var result = sort(testarray2);
36 console.log(result);
```

Bibliography

Bruce Childers, J.W. Davidson, and M.L. Soffa. Continuous compilation: a new approach to aggressive and adaptive code transformation. In *Proceedings International Parallel and Distributed Processing Symposium*, volume 00, page 10. IEEE Comput. Soc, 2003. ISBN 0-7695-1926-1. doi: 10.1109/IPDPS.2003.1213375.

William Choi. Rehearse: Coding Interactively while Prototyping. Technical report, Stanford University, 2008.

William Choi, Joel Brandt, and Scott R. Klemmer. Rehearse: Coding Interactively while Prototyping. *UIST'08*, October 19, 2008.

Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN 0596517742.

Jonathan Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39(12):84, December 2004. ISSN 03621340. doi: 10.1145/1052883.1052894.

David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 281–292, 2003. doi: 10.1109/ISSRE.2003.1251050.

David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*, page 76, 2004. doi: 10.1145/1007512.1007523.

Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental Compilation in Magpie. *ACM SIGPLAN Notices*, 19(6):122–131, 1984.

Index

Typeset September 26, 2013