Concurrent and asynchronous JavaScript programming using Reo

Thesis
Concurrent and asynchronous JavaScript programming using Reo

Thesis
Abstract

WebWorkers and other asynchronous JavaScript APIs (like XHR/AJAX and GeoLocation) provide us a facility to process (large) tasks in the background to ensure that a JavaScript application stays responsive and does not freeze up.

The way these asynchronous APIs are exposed to the application logic is usually via events or callbacks. Unfortunately, events and callbacks induce a non-linear control flow and (as a result) JavaScript programs may become difficult to understand, maintain and test. WebWorkers, for instance, utilize asynchronous message passing to communicate with the main thread. Designing and implementing these message-passing contracts and writing event listeners (to act upon those messages) is time-consuming and error-prone.

One approach to improve the problems related to asynchronous/concurrent JavaScript programming is to separate programming of computation code (processes) from programming of synchronization/communication code (protocols). For the former, programmers can perfectly use JavaScript, but for the latter (which is hard to do in JavaScript), programmers can use a special domain-specific language (DSL), specifically designed to make programming of protocols simpler. Reo is such a DSL.

In this thesis we use Reo as a ‘glue language’ for modelling the synchronization/communication logic between asynchronous JavaScript APIs (like WebWorkers, XHR/AJAX and GeoLocation) and the main thread. The visual representation of Reo gives better insight in the data flow of an asynchronous/concurrent JavaScript application. Isolating synchronization/communication logic leads to reusable, pure coordination code (protocols), while programmers can focus on writing computation code (processes) [1].

We demonstrate that it is actually possible to write concurrent/asynchronous JavaScript programs using Reo by implementing two reference applications: One application that communicates with WebWorkers and another application orchestrating multiple asynchronous JavaScript APIs. We also demonstrate that a Reo application can run both client-side (in a web browser) and server-side (in Node.js [19]).

We conclude that Reo gives some nice advantages related to separation of responsibilities (between processes and protocols), re-usage of processes/protocols and understanding the control-flow of a JavaScript application. A disadvantage is that Reo comes with a learning curve. Alternatives discussed in related work are sometimes easier to apply for a specific problem.

Our suggestions for further research include researching if Reo can improve the testability of JavaScript applications by model checking. Additionally, we should improve the Reo designer to improve usability and decrease Reo’s learning curve for JavaScript developers.
6 Implementing processes

6.1 Implement processes using the API for ports
   6.1.1 Implement process as JavaScript function
   6.1.2 Implement process as JavaScript class

6.2 Why blocking I/O operations are impossible in JavaScript

6.3 Synchronous looking process code using the API for ports
   6.3.1 Promises
   6.3.2 Async functions
   6.3.3 Generator functions
   6.3.4 JavaScript compatible alternatives
   6.3.5 Reo transpiler

6.4 Implement processes as an asynchronous JavaScript API method

7 Link Reo circuit to JavaScript processes

7.1 Link processes implemented using the API for ports
   7.1.1 Link processes to run in a background thread (Web Worker)
   7.1.2 Link processes to run in the main thread (protocol context)

7.2 Link asynchronous JavaScript APIs to a Reo protocol
   7.2.1 Link hand-written promise-based API method
   7.2.2 Link hand-written callback-based API method
   7.2.3 Link external asynchronous JavaScript APIs (like jQuery)

8 Generating code

8.1 The Reo-to-JavaScript compiler

8.2 Compiler generated Workers invoking the linked processes
   8.2.1 Processes running in a background thread (Web Worker)
   8.2.2 Processes running in the main thread (protocol context)

8.3 Compiler generated JavaScript protocols

8.4 Compiler generated JavaScript programs
   8.4.1 Run generated HTML application in a web browser
   8.4.2 Run generated Node.js application in a terminal
   8.4.3 Embed a Reo circuit in another JavaScript application

9 Validation

9.1 Reference 1: Map/reduce problem using Web Workers
   9.1.1 Designing the protocol
   9.1.2 Writing the processes
   9.1.3 Running the test application(s)
   9.1.4 Understanding the original reference application
   9.1.5 Embed the Reo circuit into the reference application
   9.1.6 Performance measurement
   9.1.7 Evaluation
1. Introduction

Historically, browsers have been single threaded, forcing all JavaScript code in an application to run in a single UI thread [20]. This limitation made it impossible to take advantage of the multi-core machines that modern systems have to provide a good user experience.

Luckily, today’s JavaScript engines provide a number of asynchronous APIs such as the commonly used XHR (XMLHttpRequest or ‘AJAX’) API, as well as IndexedDB, SQLite and the HTML5 GeoLocation API [21]. With the HTML5 Web Workers API [22] developers can spawn and implement their own background threads overcoming previous single threaded limitations. Nowadays JavaScript is even used to write server-side code (e.g. using Node.js) [23].

However, one of the most difficult aspects of building asynchronous JavaScript applications is the interaction and synchronization between these asynchronous browser APIs. The way asynchronous programming is exposed to the application logic is via events or callbacks [21]. For example, the simple (and imaginary) ‘getData()’ API, showing a message-box as soon the result is available, would become something like this:

```javascript
getData(function(data) {
  alert("We got data: " + data);
});
```

Listing 1.1: JavaScript callback function

One limitation of the above callback approach is that it can become really cumbersome to write even moderately advanced synchronization logic [21]. For example, if we need to wait for two asynchronous API calls to finish before making a third one, code complexity can rise quickly (as described in Section 2.2). Luckily, JavaScript is evolving so we can write asynchronous programs that look more like synchronous programs [15]. Promises and generators have arrived with ECMAScript version 6, and ‘async’ and ‘await’ keywords are coming in a future ECMAScript version.

Despite these new JavaScript language features, working with asynchronous code can still be very complex, especially for programmers only accustomed to synchronous APIs [24]. The reason it is difficult for developers to write asynchronous code is that most people think in step-by-step terms, but it is hard to express asynchronous code in a step-by-step fashion [16]. Section 2.2 and Section 6.3 gives a closer look at these challenges.

Because interaction- and synchronization logic is often intertwined with application-specific logic, it is difficult to reuse this logic in another JavaScript application. It is difficult to isolate this coordination code. As a result, writing, testing, and reasoning about interaction- and synchronization logic is difficult and time consuming.

1.1 Contribution

This research is about decreasing the complexity of concurrent and asynchronous JavaScript programming by using Reo [2]. Reo is a domain-specific language (DSL) specifically designed for programming synchronization/communication code. Using Reo we separate programming of computation code from programming of synchronization/communication code. For the former, programmers can perfectly use JavaScript, but for the latter (which is hard to do in JavaScript), Reo might be a better option.

Reo is a graphical coordination language defining the dataflow among processes [2]. The definition of the dataflow is called the protocol. In the world of JavaScript, these processes
will either run in a background thread (in a HTML5 Web Worker) or asynchronously in one of the built-in browser APIs (like the HTML5 GeoLocation API).

In this thesis we will use Reo as a ‘glue language’ for the orchestration of Web Workers and (other) asynchronous browser APIs. The visual representation of a Reo circuit should give better insight in the data flow of a JavaScript application. The concept of exogenous coordination (“coordination from outside” [2]) leads to reusable, pure coordination code, while programmers can focus on writing computation code [1] (running in a HTML5 Web Worker). This enables a programmer to reuse a single interaction protocol in multiple JavaScript applications, because the code defining the dataflow (the protocol) is not longer intertwined with application-specific logic (the processes). With Reo it is easy to isolate the coordination code.

In this thesis, we extend Reo with support for JavaScript as a target language. To demonstrate that Reo can actually be used to support concurrent and asynchronous JavaScript programming, we tried to replace the synchronization/communication logic from some real world reference applications.

1.2 Thesis overview

The thesis starts by first describing some background information about JavaScript and Reo. Chapter 2 introduces the (concurrency model of) JavaScript. Chapter 3 is an introduction to exogenous coordination and Reo. Chapter 4 elaborates the contents of this research by highlighting the issues involving concurrent and asynchronous programming in JavaScript and proposing Reo as an possible solution to these issues. This chapter will also describe the research questions and defines a roadmap how this thesis will answer the research questions.

The main goal of this research is to extend Reo with support for JavaScript as a target language. The Reo runtime library for JavaScript is described in Chapter 5. Chapter 6 describes how a programmer can implement processes in JavaScript and Chapter 7 describes how to link these processes to a Reo circuit. Chapter 8 describes how to compile a Reo circuit linked to JavaScript processes to a (runnable) JavaScript application.

In Chapter 9 describes how the results of this research where validated. Chapter 10 gives the conclusion of this research, including answers to the research questions and an overview of related- and future work.
2. Background: JavaScript

JavaScript is a lightweight, interpreted, programming language with first-class functions. Most well-known as the scripting language for Web pages, many non-browser environments use it such as Node.js and Apache CouchDB \[23\]. JavaScript is a prototype-based, multi-paradigm, dynamic scripting language, supporting object-oriented, imperative, and functional programming styles \[25\]. The core features of JavaScript are based on the ECMA-Script standard \[23\], but JavaScript also has other additional features that are not in the ECMA specifications/standard.

Contrary to popular misconception, JavaScript is not a lightweight version of Java \[23\]. The basic syntax is intentionally similar to both Java and C++ to reduce the number of new concepts required to learn the language \[25\]. Language constructs, such as if statements, for and while loops, and switch and try/catch blocks function the same as in these languages (or nearly so).

Most JavaScript-applications perform actions as a response to HTML events \[26\]. An event is a signal from the browser that something (for example a mouse click) has happened. There should be an event-handler function assigned to an event to react to this particular event.

Imagine we have a simple HTML page with a button. In JavaScript, the button is represented as an object with an ‘onclick’ event. This event holds a pointer to the JavaScript function that is executed once the button is clicked. Listing 2.1 assigns an event-handler to the ‘onclick’ event of a button.

```javascript
button.onclick = function() {
    // myScript
};
```

Listing 2.1: Assigning an event-handler to the ‘onclick’ event of a button

2.1 JavaScript concurrency model and Event Loop

JavaScript event handling is single-threaded \[26\], so handlers are executed sequentially. This means that if two events happen simultaneously, their handlers will be executed one after another.

JavaScript has a concurrency model based on an ‘event loop’. This model is quite different than the model in other languages like C or Java. A JavaScript runtime contains a message queue, which is a list of messages to be processed. Each message is associated to a function. When the runtime is idle, a message is taken out of the queue and processed by the main thread. This involves calling the associated function \[27\].

Each message is processed completely before any other message is processed in a single threaded fashion. This offers some nice properties when reasoning about a program, including the fact that whenever a function runs, it cannot be pre-empted and will run entirely before any other code runs (and can modify data the function manipulates). A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the “a script is taking too long to run” dialog as shown in Figure 2.1 \[27\]. When we stop executing the script, the function on that web page that is dependent upon the script might not function properly.
Modern browsers provide a number of asynchronous APIs for I/O operations \[21\]. These operations (XHR, Web Workers, etc.) are executed within a Thread Pool. The threads in this Thread Pool are background threads controlled by the browser running independent from the main thread. When a function calls an asynchronous API function, the main thread does not block, but will continue processing \[27\]. As soon as the asynchronous operation returns, a new message associated to a callback or event-handler function is placed into the message queue. This callback function is responsible for processing the result of the asynchronous function and will be executed by the event loop in a single threaded fashion.

### 2.1.1 Events

Events are commonly used for asynchronous operations that should notify the caller about its completion \[27\]. Listing 2.2 shows an example for the XHR (XMLHttpRequest or ‘AJAX’) API. On the first line, an `XMLHttpRequest` object is created. This object represents the HTTP request to be made. On the second line, a connection to a particular URL is opened, using an HTTP-get action (but the request is not yet actually made). On the third line, the event-handler for the eventual response of the request is initialized. This means that once a response from the remote server at location ‘some/url/1’ is received, a message associated to the `handleResponse` function is appended to the message queue. Finally, on the fourth line, the HTTP request is actually made; the meaning of the null argument does not matter in this example. The actual HTTP call is executed in a background thread, but the event-handler function ‘handleResponse’ is executed in the main thread.
```javascript
var xmlhttp = new XMLHttpRequest();
xhttp.open("GET", "some/url/1", true);
xhttp.onreadystatechange = handleResponse;
xhttp.send(null);

function handleResponse(data) {
    if (xmlhttp.readyState === 4) {
        console.log(data);
    }
}
```

**Listing 2.2: Events**

### 2.1.2 Callbacks

A callback function is a function passed as a parameter to another function and is commonly used to notify the caller about the completion of an asynchronous operation [27]. For example the HTML5 Geolocation API adds a message associated to the ‘showPosition’ callback function to the message queue when user’s position is available. The logic to obtain the current position is executed in a background thread, but the function ‘showPosition’ function is executed in the main thread.

```javascript
navigator.geolocation.getCurrentPosition(showPosition);

function showPosition(position) {
    console.log(position);
}
```

**Listing 2.3: Callbacks**

A common approach is to wrap event-based APIs into a callback-based API to define a generic API with a set of common and reusable functions. Listing 2.4 shows how to create a reusable HTTP-get function (called ‘httpGet’) based on the event-based XHR (XMLHttpRequest or ‘AJAX’) API. Instead of using the low level ‘XMLHttpRequest’ object (as described in Section 2.1.1) over and over again, we can just use the ‘httpGet’ function with the url as parameter.

```javascript
function httpGet(url, callback) {
    var xmlhttp = new XMLHttpRequest();
xhttp.open("GET", url, true);
xhttp.onreadystatechange = function(data) {
    if (xmlhttp.readyState === 4) {
        callback(data);
    }
};
xhttp.send(null);

httpGet("some/url/1", function(data) {
    console.log(data);
});
```

**Listing 2.4: Wrap event-based XHR API into a callback-based API**

### 2.2 Synchronization logic: Avoid the callback hell

One limitation of events and callbacks is that it can become really cumbersome to write even moderately advanced synchronization logic [21].
For example, if we need to wait for two asynchronous API calls (‘callA’ and ‘callB’) to be done before doing a third one (‘callC’), code complexity can rise quickly. Listing 2.5 shows how our code would become if we want to implement this behaviour using callbacks. First, the API call ‘callA’ is invoked. Subsequently, the API call ‘callB’ is invoked. Because we do not know which API call is completed first, the callback function of both API calls (‘callA’ and ‘callB’) should check if the other API call is also completed. Only when the result of the other API call is available, the API call ‘callC’ is invoked.

```javascript
var resultA, resultB;  // We need to await two calls (A and B), before calling C
API.callA(function(result) {
    resultA = result;
    if (resultB) {  // Execute C when both A and B are ready
        API.callC([resultA, resultB], function(resultC) {});
    }
});
API.callB(function(result) {
    resultB = result;
    if (resultA) {  // Execute C when both A and B are ready
        API.callC([resultA, resultB] function(resultC) {});
    }
});
```

Listing 2.5: Orchestration of asynchronous APIs using callbacks

The structure of nested callback functions is known as “callback hell” [3] or “pyramid of doom” [4], named to the pyramid-like shape of the code.

### 2.2.1 Promises

The promise API (arrived in ECMAscript 6) can be seen as a standard, generic and more scalable approach to work with callbacks [28]. The callbacks are still there, but promises force us to keep them clean. A promise represents the eventual result of an asynchronous operation which can be obtained by calling the ‘then’ function [29]. A promise is also referred to as a thenable, which is really just an object that defines a then function. The ‘then’ functions takes two arguments, one function for success, one (optional) function for failure (or ‘resolve’ and ‘reject’, in promises terms).

It is possible to queue multiple asynchronous operations and obtain the result together, making synchronization much easier.

Listing 2.6 shows how our code would become if we want to implement the same behaviour as in the previous example of Listing 2.5 using promises. Both API calls ‘callA’ and ‘callB’ return a promise. The function ‘Promise.all’ also returns a promise which will resolve when all promises passed as argument (‘callA’ and ‘callB’) are resolved. The promise returned by the function ‘Promise.all’ ensures both API calls (‘callA’ and ‘callB’) are completed and resolves with an array ‘resultAB’ containing the return values of both API calls. The API call ‘callC’ can be invoked directly in the the ‘then’ function, because we already known both depending API calls are completed.

```javascript
Promise.all([API.callA(), API.callB()]).then(function(resultAB) {
    // Execute C when both A and B are ready
    API.callC(resultAB).then(function(resultC) {});
});
```

Listing 2.6: Orchestration of asynchronous APIs using promises
2.2.2 Async functions

Async functions (identified by the `async` keyword) are functions that will execute asynchronously and will be introduced in a future ECMAScript version - and currently only available using a transpiler like babel [30]. With async functions, it is possible to await on a promise (using the `await` keyword). The executing thread stops executing the function and proceeds with executing other functions from the event queue. The halted function is resumed after the promise is resolved.

Listing 2.7 shows how our code would become if we want to implement the same behaviour as in the previous example of listing 2.6 using `async` and `await`. The `await` keyword allows us to await the `Promise.all` function without calling the `then` function. The result of both API calls (`callA` and `callB`) is available in the array `resultAB` and API call `callC` can be invoked directly.

```javascript
async function main () {
    var resultAB = await Promise.all([API.callA(), API.callB()]);
    var resultC = await API.callC(resultAB);
}
```

Listing 2.7: Orchestration of asynchronous APIs using `async` and `await`.

2.3 Web Workers

As described before, JavaScript is a single-threaded environment, meaning multiple scripts cannot run at the same time. A downside of this model is that if a particular task takes a long time to complete everything else is held up until that task finishes. This is known as blocking. In the world of client-side JavaScript applications, using a single-threaded architecture can lead to an app becoming slow or even completely unresponsive. The HTML5 Web Workers API [22] offers a solution to this problem by giving developers a way of instructing the browser to process large tasks in the background. Web Workers provide a facility for creating new threads for executing JavaScript code in. Creating new threads for handling large tasks allows us to ensure that a JavaScript app stays responsive and does not freeze up.

A Web Worker includes a separate message queue, event loop, and memory space independent from the main thread that instantiated it. Communication between the worker and the main thread is done via message passing, which looks very much like the traditional, evented code-examples we have already seen.

Creating a new worker is simple [31]. All we need to do is call the `Worker()` constructor. Sending messages to the worker is done by calling the `postMessage` function. Listening to messages from the worker is done by listening to the `onmessage` event. The sample of Listing 2.8 creates two workers to calculate the result of the formula `y = x^2 + 2`. The first worker is responsible for calculating the square of `x` and the second worker is responsible for doubling the result.
```javascript
var squareWorker = new Worker("calculator.js"), // Create worker to square
doubleWorker = new Worker("calculator.js"), // Create worker to double

// Send message to square worker to calculate square of x
squareWorker.postMessage({ method: "square", value: x });

squareWorker.onmessage = function(e) {
  // Result received from square worker
  doubleWorker.postMessage({ method: "double", value: e.data.result });
}

doubleWorker.onmessage = function(e) {
  console.log(x + "^2 * 2 = " + e.data.result); // 4^2 * 2 = 32
}
```

Listing 2.8: Creating and communicating with Web Workers

In the worker, we can respond to messages from the main thread by listening to the `onmessage` event. Sending messages back to the main thread is done by calling the `postMessage` function. Listing 2.9 shows the implementation of `calculator.js` running in the two created background threads.

```javascript
onmessage = function(e) {
  // Message received from main thread
  if (e.data.method == "square") {
    // Send the squared value to the main thread
    postMessage({ result: e.data.value * e.data.value });
  } else if (e.data.method == "double") {
    // Send the doubled value to the main thread
    postMessage({ result: 2 * e.data.value });
  }

  });
```

Listing 2.9: Implementation of a Web Worker (`calculator.js`)”

Sequence diagram 2.3 shows the communication flow between the main UI thread and the two workers.

![Sequence diagram 2.3](image)

Figure 2.3: Communication between the Main UI thread and web workers

Web Workers are relatively heavy-weight, and are not intended to be used in large numbers [32]. In Firefox, for example, a Web Worker runs in a separate runtime with its own stack, heap, and message queue [27]. Compared to threads in Java or C, Web Workers have a
high start-up performance cost, and a high per-instance memory cost.

2.4 ECMAScript6

ECMAScript (abbreviated as ES) is the standardized scripting language that JavaScript (and some other languages, like ActionScript) implements. ECMAScript 6, also known as ECMAScript 2015, is the latest version of the ECMAScript standard. ES6 is a significant update to the language, and the first update to the language since ES5 was standardized in 2009. ES6 is finalized, but not yet fully supported by all browsers \[33\]. It is possible to use an ES6 transpiler (like Babel \[34\]) that will convert ES6 code to ES5-compatible code supported by all browsers.

ECMAScript 6 has some really useful features. We already lifted a corner of the veil by describing the Promises in Section \[2.2.1\]. Besides this, everything discussed so far is in ES5. These (other) features which are used by the Reo runtime for JavaScript will be described in this section.

2.4.1 JavaScript classes

JavaScript classes are introduced in ECMAScript 6 and are syntactic sugar over JavaScript’s existing prototype-based inheritance. With these new syntax elements it is possible to define classes in JavaScript with similar syntax as Java. For example, an implementation of an ‘Animal’ class given in Listing 2.10.

```javascript
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(this.name + ' makes a noise.');
    }
}
```

Listing 2.10: Class declarations

The ‘extends’ keyword is used in class declarations to create a class as a child of another class. For example, Listing 2.11 gives an implementation of a ‘Dog’ class extending the ‘Animal’ class.

```javascript
class Dog extends Animal {
    speak() {
        console.log(this.name + ' barks.');
    }
}
```

Listing 2.11: Sub classing with extends

2.4.2 Modules

ES6 introduces a standardized module format to JavaScript. Each module is defined in its own file. The functions, objects or primitives defined in a module are not visible outside this module unless we explicitly export them using the ‘export’ keyword. The ‘import’ keyword is used to import functions, objects or primitives that have been exported from an external module. In the following example the class ‘Animal’ is exported from the module ‘Animal.js’ and imported in the module ‘Dog.js’.

9
At this moment, no JavaScript engine supports modules natively. Getting everything working in current browsers requires a bundle step. Bundling modules means combining several files with modules into a single JavaScript file. There are a few popular bundlers like Browserify [35], Webpack [36] or Rollup [37].

### 2.4.3 Generators

ECMAScript 6 introduces the ‘yield’ keyword. Anyone with some programming experience in C# or Python (and a number of other languages) may already be familiar with how ‘yield’ can suspend execution of a function and return control (and a value) to the caller. At some later point, execution can return to the point just after yield occurred. In ES6, functions using the yield keyword are known as generator functions and have a special syntax ('function*').

Listing 2.14 demonstrates the usage of a generator function. The ‘yield’ operation on line 2, 3 and 5 suspends the execution of the ‘numberGenerator’ function and return control (and a value) to the caller (the body of the for-loop on line 13). After the yielded value printed the execution of ‘numberGenerator’ is continued. As a result the operation ‘console.log("About to go 10")’ is not executed immediately.

```javascript
function* numberGenerator() {
  yield 1;
  yield 2;
  console.log("About to go to 10");
  yield 10;
};

var iterator = numberGenerator();
var next = iterator.next();

//The following code will print: 1, 2, About to go to 10, 10
while(!next.done) {
  console.log(next.value);
  next = iterator.next();
};
```

Listing 2.14: Usage of generator functions
3. Background: Reo coordination language

This chapter describes Reo as part of the PrDK development kit. PrDK is a collection of plugins for Eclipse enabling programmers to write concurrent programs using Java as a language for writing processes and Reo for specifying protocols. Section 3.1 is a brief introduction of Reo [2]: a DSL for writing concurrency protocols. Section 3.2 focuses on concurrent programming in Java using the PrDK Development Kit [5] containing Reo. A Reo specification can be compiled into a working Java program. Section 3.3 describes how the Reo-to-Java compiler works and how the generated code looks. The runtime behaviour of the generated protocol is described in Section 3.4. To conclude, Section 3.5 describes some different compilation approaches and optimization techniques.

3.1 Reo: a DSLs for writing concurrency protocols

Conceptually, concurrent programs consist of processes, which implement primary modules of sequential computation, and protocols, which implement the rules of concurrent interaction that processes must abide by [5]. Today’s popular programming languages like Java, C, C# and even JavaScript have suitable constructs and abstractions for writing sequential code and implementing processes. However these programming languages do not provide explicit, high-level elements of syntax for programming protocols [5]. Instead, programmers need to use rather low-level constructs like shared memory, mutual exclusion, message passing, events, etc.

This research is inspired by a long-term project at CWI (Centrum Wiskunde & Informatica, Amsterdam), studying an alternative approach to concurrent programming, based on syntactic separation of processes from protocols. In this approach, programmers write their (sequential) processes in a GPL (general purpose language) like Java, while they write their (concurrency) protocols in a DSL (domain-specific language) [5]. DSLs are small languages, focused on a particular aspect (domain) of a software system (e.g. concurrency protocols in this particular case) [17].

3.1.1 Reo coordination language

Reo is a premier example of a DSL to describe (concurrency) protocols [1]. Reo has a graphical syntax in which every protocol (called a circuit), is a labeled directed graph. Such a graph represents the data-flow among the processes in the system. Edges in the graph (called channels) are labeled by their channel type (graphically represented as a different edge shape). The type of a channel affects the data flow through that channel. Reo has the following primitive channel types [1]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Sync</strong> channel has a source (A) and a sink end (B) and no buffer. It accepts a data item through its source end (A) iff it can synchronously dispense it through its sink (B).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>LossySync</strong> channel is similar to a Sync channel except that it always accepts all data items through its source end (A). This channel loses the data item only whenever it cannot synchronously dispense it through its sink (B).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>FIFO1</strong> channel represents an asynchronous channel with a buffer of capacity 1: it can contain at most one data item.</td>
</tr>
</tbody>
</table>
A **SyncDrain** channel is a synchronous drain that accepts a data item through one of its ends iff it synchronously accepts a data item through its other end as well.

An **AsyncDrain** channel is an asynchronous drain that accepts data items through exactly one of its source ends at a time and loses that data item.

A **Filter(P)** channel is similar to a Sync channel except that it loses data items that do not match pattern P.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="image" /></td>
<td><img src="image2" alt="image" /></td>
</tr>
</tbody>
</table>

A **source node** has only coincident source ends. A process can write data items to a source node that it is connected to. This operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a **synchronous replicator**.

| ![image](image3) |

A **sink node** has only coincident sink ends. A process can obtain data items from a sink node that it is connected to. This operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a data item; if more than one coincident channel end offers data items, one is selected nondeterministically. A sink node, thus, acts as a **nondeterministic merger**.

| ![image](image4) |

A **mixed node** has both coincident source and coincident sink ends. A circuit uses its mixed nodes only for internally routing data. A mixed node nondeterministically selects and takes a data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

<table>
<thead>
<tr>
<th>A</th>
<th>P</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5" alt="image" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Primitive channel types in Reo

Channels have ends and are joined together on nodes. A node is a logical place where channel ends coincide and coordinate their data-flows as prescribed by its node type [1].

| ![image](image6) |

A **source node** has only coincident source ends. A process can write data items to a source node that it is connected to. This operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a **synchronous replicator**.

| ![image](image7) |

A **sink node** has only coincident sink ends. A process can obtain data items from a sink node that it is connected to. This operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a data item; if more than one coincident channel end offers data items, one is selected nondeterministically. A sink node, thus, acts as a **nondeterministic merger**.

| ![image](image8) |

A **mixed node** has both coincident source and coincident sink ends. A circuit uses its mixed nodes only for internally routing data. A mixed node nondeterministically selects and takes a data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

<table>
<thead>
<tr>
<th>A</th>
<th>P</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image9" alt="image" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Node types in Reo

The source nodes and sink nodes of a circuit constitute its set of **boundary nodes**. Output ports (input ports) of a process are connected to the source nodes (sink nodes) of a Reo circuit.

Every process owns a set of **ports** [2] permitting a process to perform I/O operations on the boundary nodes of the circuit to which they are connected. **Output ports** let processes offer data to the circuit, while **input ports** let processes accept data from the circuit [5]. All I/O operations are blocking [5], which means that a process can proceed only after its pending I/O operation has been successfully processed. When a process performs a ‘put’ (‘get’) on an output port (input port), this operation becomes pending on that port and the process itself becomes suspended. When a ‘put’ (‘get’) completes, its previously suspended process resumes and offers (accepts) a data item [5].

We introduce a producers/consumer protocol named ‘**LateAsyncMerger2**’ [6] involving two producers (A and B) and one consumer (C). Initially, a ‘put’ by ‘Producer A’ can
complete, causing that producer to offer a data item into the circuit. Alternatively, a ‘put’ by ‘Producer B’ can similarly complete. Subsequently, only a ‘get’ by ‘Consumer C’ can complete, causing the consumer to accept the data item from the circuit. This protocol, thus, admits asynchronous (via a FIFO channel), unordered, reliable, transactional (i.e., different communications of data do not interleave) communication from two producers to a consumer.

The Reo diagram (Figure 3.1) implementing this protocol identifies the following concepts:
Three processes (‘ProducerA’, ‘ProducerB’ and ‘ConsumerC’), a circuit (‘LateAsyncMerger2’), two source nodes (‘A’ and ‘B’), a sink node (‘C’), an internal mixed node (‘Z’), two Sync channels, a FIFO1 channel, two output ports (‘Aout’ and ‘Bout’) and one input port (‘Cin’).

3.1.2 Pr: the textual superset of Reo

Often, programmers need different versions of a program with different numbers of processes. The Reo syntax does not conveniently support this. For instance, Reo requires programmers to draw a specific diagram for a protocol among two producers, another specific diagram for the same protocol among three producers, etc. Reo does not support drawing a generic diagram for any (nonzero) number (N) of producers and one consumer. Pr, the textual superset of Reo [5], does support such parametrization.

Programmers can use the Reo-to-Pr translator to translate their Reo diagram into a Pr text, which they subsequently can modify by parameterizing the protocol [5]. The example below modifies the ‘LateAsyncMerger2’ protocol described in the Section 3.1.1 to a ‘LateAsyncMergerN’ protocol. Instead of the two producers A and B this protocol has N producers (A[1...N]).

![LateAsyncMerger2](image)

**Figure 3.1:** Reo circuit for the ‘LateAsyncMerger2’ protocol

Listing 3.1: ‘LateAsyncMerger2’ with 2 producers (‘A’ and ‘B’) and one consumer (‘C’)

```
LateAsyncMerger2(A,B;C) =
    Node ([A];[p1])  // Name: A
    mult Node ([B];[p2])  // Name: B
    mult Sync (p1;p5) // A to X
    mult Sync (p2;p6) // B to X
    mult Node ([p5 , p6];[p4])  // Name: X
    mult Fifo (p4;p3) // X to C
    mult Node ([p3];[C])  // Name: C
main = LateAsyncMerger2(A,B;C)
```

Listing 3.2: ‘LateAsyncMergerN’ with N producers (‘A1’...’AN’) and one consumer (‘C’)

```
LateAsyncMergerN(A[]; C) =
    prodi:1..#A {
        Node ([A[i]];[p1[i]])  // Name: A[1...N]
        mult Sync (p1[i];p5[i]) // A[i] to X
    }
    mult Node (p5[1...#A];[p4])  // Name: X
    mult Fifo (p4;p3) // X to C
    mult Node ([p3];[C])  // Name: C
N = 3
main = LateAsyncMergerN(A[1...N]; C)
```
The details of Pr do not matter in this thesis; we refer the interested reader to [5].

3.1.3 Constraint automata: the formal basis of Reo

By effectuating only admissible interactions, protocols (described in Reo or Pr) essentially constrain the completion of ‘put’/‘get’ operations [5]. Formally, we can represent such constraints with an automaton [5]. An automaton consists of a number of states and transitions between states. Each automaton has an initial- and a current state. A synchronization constraint and a data constraint is attached to each transition between states, specifying the conditions under which a transaction can fire. A state can have local memory variables to store data (e.g. the contents of a FIFO channel). We call the type of automatons described in this section constraint automata [7].

The following table shows the automata semantics for the common set of Reo primitives described in Section 3.1.1 [1].

<table>
<thead>
<tr>
<th>Automaton</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>The Sync automaton has a single state and a single transition. The synchronization constraint {A, B} states that this transition is possible iff both nodes A and B can fire synchronously, allowing their respective pending I/O operations to succeed. The data constraint (d(A) = d(B)) states that this transition is possible iff the data observed at node A is identical to the data observed at node B. Because these two nodes are respectively the source and the sink nodes (of the Sync channel), this data constraint requires a transfer of data from A to B.</td>
</tr>
<tr>
<td>LossySync</td>
<td>The LossySync automaton has a single state and two transitions. One of these transitions is identical to that of the Sync channel, modeling its identical behavior. The other, labeled by A, true simply states that the automaton can make this transition iff A can fire by itself and imposes no constraint of the data of A: this data is lost.</td>
</tr>
<tr>
<td>FIFO1</td>
<td>The FIFO1 automaton has two states, representing its empty (initial) and full states. The label {A}, (d(A) = X') of the transition that takes the automaton from its empty to its full state allows it to make this transition iff node A can fire by itself, and the new value of the memory variable X in the target state (identified by X’ in the data constraint) is the same as the data value observed on node A: the value obtained from the source node A gets assigned to the X variable of the target state to satisfy this constraint. The label {B}, (d(B) = X) of the transition that takes the automaton from its full to its empty state allows it to make this transition iff node B can fire by itself, and the value of the memory variable X in the source state (identified by X in the data constraint) is the same as the data value observed on node B: the value of the X variable of the source state is dispensed through the sink node B to satisfy this data constraint.</td>
</tr>
</tbody>
</table>
The **SyncDrain** automaton has a single state and a single transition, whose constraints require its ends to fire synchronously (A, B), but imposes no constraints (true) on their data. Because these are both source ends, their data are simply lost.

The **AsyncDrain** automaton has a single state and two transitions, each of which allow it to fire and lose the data obtained through one of its ends (but never both / synchronously).

The **Filter(P)** automaton has a single state and two transitions. If source node A can fire and its data value does not match the filter pattern P, then the data value of A is simply lost. If the data value available on the source node A matches the filter pattern P, then the only possible transition is one similar to that of the Sync channel, by which the data value of A is transferred to the sink node B.

The **Node** automaton has a single state and a transition per coincident sink end. This example has two coincident sink ends (A and B) and one coincident source end (C). The transition \{A, C\}, d(A)=d(C) is possible iff both nodes A and C can fire synchronously, allowing their respective pending I/O operations to succeed. The data constraint d(A) = d(C) requires a transfer of data from A to C. The transition \{B, C\}, d(B)=d(C) works similar. The total number of transitions depends on the number of coincident source- and sink ends.

Using a special multiplication operation (which represents parallel composition), the primitive automata in Table 3.3 (each of which represents a simple protocol) can be multiplied into compound ones (each of which represents a more complex protocol) [5]. The details of the multiplication do not matter in this thesis. Reo and Pr are two declarative syntaxes for representing such multiplication expressions [5]. Given such a set of primitives, in Reo, programmers draw multiplication expressions as dataflow graphs; in Pr, programmers write multiplication expressions as automata signatures [5]. In other words, Reo and Pr hide the difficulties of automata from programmers and offer an easier and more scalable approach for defining automata based on their (parallel) composition. The semantics for an arbitrary Reo circuit (or Pr text) can be compositionally computed by forming the **product** (multiplication) of the ‘small’ automata for that circuits constituents (i.e., nodes and channels).

The example below shows the constraint automaton for the ‘**LateAsyncMerger2**’ protocol described in Section 3.1.1 [5]. This example ‘multiplies’ two Sync automata with a FIFO1 automaton and four node automata (for A, B, C, and Z).
Contrasting their constraint automata, Reo circuits (or Pr texts) will not grow prohibitively large when multiple FIFO-channels are used (multiplied) in a circuit. To illustrate this scalability advantage, consider the protocol ‘EarlyAsyncMerger2’ (see Figure 3.3) which is just a slightly modified version of the ‘LateAsyncMerger2’ protocol discussed before. The modified protocol states that the producers send their data to the consumer asynchronously, reliably, unordered, but non-transactional (in contrast to ‘LateAsyncMerger2’, different communications of data may interleave) [5]. The constraint automaton for ‘EarlyAsyncMerger2’ shows a beginning ‘state explosion’. Generally, the constraint automaton for k producers has as many as 2^k states (where k equals the number of FIFO channels) [5]. Contrasting their constraint automata, the Reo graphs for ‘EarlyAsyncMerger2’ grow only linearly in k (instead of exponentially), while its Pr texts even stay constant.

3.2 The PrDK development kit

The Pr development kit, called PrDK, consists of tools (Eclipse plugins) for protocol programming with automata, without ever exposing programmers to automata directly [5]. PrDK consist of editors for Reo and Pr, an animation engine for Reo, a Reo-to-Pr translator, a parser/interpreter and a Pr-to-Java compiler [5].
In the basic workflow, the Reo-to-Pr translator, the parser/interpreter and the Pr-to-Java compiler are transparently chained together, giving the programmer the illusion of a Reo-to-Java compiler [5]. The basic workflow can, thus, be extended with an extra step in which programmers explicitly use the Reo-to-Pr translator to translate their Reo diagram into a Pr text, which they subsequently can modify and parameterize as described in Section 3.1.2 [5]. Figure 3.4 illustrates the basic programming workflow:

![Figure 3.4: Programming workflow with PrDK](image)

### 3.2.1 Implement Java processes using the API for ports

PrDK comes with a runtime library for Java. Processes can be implemented easily by using the API for ports: `OutputPort` for producers and `InputPort` for consumers. The interface description is very simple: An `InputPort` with a `get` function and an `OutputPort` with a `put` function. Listing 3.3 shows some hand-written processes for the producer/consumer example.

```java
public class Processes {
    public static void Producer(OutputPort p, int id) {
        String message = id + " : Hello, World!";
        while (true) {
            p.put(message);
        }
    }

    public static void Consumer(InputPort p) {
        while (true) {
            System.out.println(p.get());
        }
    }
}
```

Listing 3.3: Hand-written processes in Java

In Section 3.2.2 and 3.2.3 we show how to link these static methods as processes to the `LateAsyncMerger2` protocol described in Section 3.1.1.

### 3.2.2 Draw protocols using the graphical Reo editor

Programmers can draw a protocol as a Reo circuit by using the graphical user interface of the Reo editor. The animation engine enables programmers to visualize the admissible data-flows through the graph, which is an instructive and helpful aid in protocol debugging [5]. Subsequently, programmers can import processes, by drag/dropping Java files onto the same canvas (which appear as boxes alongside the graph, with distinct markers for
their ports), and link (the ports of) those processes to (the nodes in the graph of) the protocol as desired [5]. The resulting diagram comprehensively implements a full program [5]. Figure 3.5 shows the ‘LateAsyncMerger2’ protocol in the Reo editor, where the processes (Listing 3.3) were imported by drag/dropping this Java file onto the canvas.

3.2.3 Write protocols using the editor for Pr

When using the textual editor for Pr, programmers write multiplication expressions as automata signatures. Figure 3.6 shows the ‘LateAsyncMerger2’ protocol in the Pr editor, where the processes are referenced directly in Pr text.

3.3 Compiler generated Java protocols

Roughly, the compiler and its generated code work as follows. First, the Pr parser/interpreter translates a Pr text to a constraint automaton. Second, the compiler translates the resulting product automaton (which comprehensively models a protocol - see Section
(3.1.3) into a Java class (which effectively encapsulates a state machine for simulating that automaton) [5].

Figure 3.7: Class diagram for the Java implementation of the ‘LateAsyncMerger2’ protocol.

The class diagram in Figure 3.7 shows the generated code for the ‘LateAsyncMerger2’ protocol described in Section 3.1.1. The constructor of the protocol class (‘LateAsyncMerger2’) has a number of formal port parameters (‘a’, ‘b’ and ‘c’), to bind its instance to actual ports. This protocol class effectively encapsulates a state machine class (‘MyAutomaton’) for simulating the automaton generated by the Pr parser/interpreter. Corresponding to the constraint automata in Figure 3.2, the generated state machine contains two states (‘State1’ and ‘State2’) and three transitions (‘Transition1’ corresponds to A, d(A) = X’, ‘Transition2’ to B, d(B) = X’ and ‘Transition3’ to C, d(C) = X’). Each transition has a target state to move the automaton from its current state to a certain target state (e.g. ‘Transition1’ moves the automaton from ‘State1’ to ‘State2’). The automaton has a reference to its current state (either ‘State1’ or ‘State2’) and contains a context representing the set of pending I/O operations. Each actual port is associated to a handler (e.g. ‘HandlerForA’ for port ‘A’) which is responsible for handling the actual I/O operations of the associated port. We explain the workings of handlers in Section 3.4. The class ‘MemoryCell1’ is used to store a local memory variable (the contents of the FIFO channel).

Figure 3.8: Class diagram for the instantiation and binding of protocols, processes and ports.
The task of constructing ports and passing them both to the constructor of a protocol class and to process methods is performed in the main method of the program. This main method is, as the protocol class, generated by the compiler (based on linkage information either in a Reo diagram or in its Pr equivalent). Figure 3.8 shows the relation between the program (with the main method), protocol and processes.

3.4 Runtime behavior of the generated Java protocol

The sequence diagram of Figure 3.9 illustrates how the generated code behaves after construction with the process ‘Producer A’ as example. When the process ‘Producer A’ performs an I/O put operation on port A, this port will call its handler (‘HandlerForA’) in an infinite loop until this operation succeeds (when all constraints are valid). The handler checks if the automaton is in the correct state (‘State1’) and tries to fire the transition ‘Transition1’. To fire a transition, the generated code first checks the synchronization constraint (confirm port A has a pending I/O operation) and the data constraint.

When the synchronization constraint is currently not satisfied, the Reo runtime ‘waits’ for another I/O operation to succeed using a binary semaphore. As soon another I/O operation succeeds, the semaphore is released and the handler-function retried within the infinite loop earlier mentioned. When all constraints are satisfied, the value of the pending I/O operation on port A is transferred to the memory cell ‘MemoryCell1’ and the reach-method of the target state (‘State2’) is called allowing this state to become active. The sequence diagram for the process ‘Producer B’ will work analogously.

The sequence diagram of Figure 3.10 illustrates the behavior of the process ‘Consumer C’. When the process ‘Consumer C’ performs an I/O operation on port C, this port will call its handler (‘HandlerForC’). The handler checks if the automaton is in the correct
state (‘State2’) and tries to fire the transition ‘Transition3’. The transition checks the synchronization constraint (confirm port C has a pending I/O operation) and the data constraint. When all constraints are satisfied, the value of the memory cell ‘MemoryCell1’ is transferred to the pending I/O operation on port C. Subsequently, the reach-method of the target state (‘State1’) is called allowing this state to become active again.

3.5 Different compilation approaches and optimization techniques

The ‘LateAsyncMerger2’ protocol described in Section 3.4 is compiled using the so called “Centralized Approach” [6] and has only a single (‘central’) automaton class. This approach fits well for automata with only a few states, but does not scale well to automata with many states because of a potential state explosion problem (as described in Section 2.1.3). ‘EarlyAsyncMerger2’ [56], for example, has more states (roughly $10^{77}$) than the observable universe has hydrogen atoms (overestimate: $10^{80}$) [6]. Generating an automaton class this large and loading this class in memory is impossible. This section describes another compiler approach called the “Distributed Approach” or “Hybrid Approach” [6] that avoid the problem of state explosion. Using this approach, a Reo circuit is split into several regions, each representing a sub-protocol. These sub-protocols are glued together using private ports. Each sub protocol has only a limited set of states which solves the state explosion problem at compile time. At runtime (‘glued together’) the overall state is depending of the state of each individual sub-protocol. The total number of runtime states is thus depending of the number of unique combinations of sub-protocol (compile time) states.

Figure 3.11 shows how the ‘EarlyAsyncMerger2’ protocol is cut into pieces:
Each sub-protocol is individually compiled into an automaton class. The ‘EarlyAsyncMerger2’ protocol will result in five automaton classes that can communicate with their neighbor automaton through private ports:

1. A node automaton from public port ‘D’ to private port ‘$inp1’;
2. A node automaton from public port ‘E’ to private port ‘$inp2’;
3. A FIFO-automaton for producer ‘D’ (activated by private port ‘$inp1’ through public port ‘D’);
4. A FIFO-automaton for producer ‘E’ (activated by private port ‘$inp2’ through public port ‘E’);
5. An automaton used by consumer ‘F’ to obtain the data from the FIFO automata (through private port ‘$out1’ or private port ‘$out2’).

Figure 3.12 shows all five automata and illustrates the private ports in between.

The difference between the Distributed- and Hybrid Approach is the partitioning method used to split the Reo circuit into regions. In the Distributed Approach each Reo channel becomes a separate automaton class. The Hybrid Approach tries to split the Reo circuit more efficiently to minimize the number of automaton classes and to reduce the communication overhead between them. More details of the Central-, Distributed- and Hybrid Approach can be found in the thesis “Automata-Theoretic Protocol Programming” [6].
4. Research design

In this chapter the contents of the research will be elaborated. Section 4.1 highlights the problems related to concurrent- and asynchronous programming in JavaScript. Section 4.2 suggests Reo as a possible solution to these problems and describes at high level what concurrent and asynchronous JavaScript programming using Reo would look like. The research questions are discussed in the Section 4.3.

4.1 Problem statement: JavaScript and its concurrency problems

In Chapter 2 we saw that programming synchronization/communication in JavaScript is complex. Coordinating data-flows in software can be cumbersome, especially when asynchronous operations finish at different times [18]. The asynchronous programming style significantly complicates program control flow and impedes program comprehension [3]. Since the business logic is split between many event handlers, the control flow is non-linear, fragmented and hard to understand. Although Promises are a positive step in eliminating the problems encountered when orchestrating a number of asynchronous callbacks, they introduce problems when used with dependent values leading to nested resolving functions [4]. Another problem is that reusing existing interaction- and synchronization logic in another application is difficult, because this code is intertwined with application specific logic. It is difficult to isolate this coordination code.

While Web Workers achieve their design goal of offloading long-running computations to background threads [8], utilizing Web Workers can be cumbersome and awkward due to the low level of abstraction. Designing and implementing message-passing contracts and writing event listeners is time-consuming and error-prone.

Another serious problem concerns the testing of asynchronous JavaScript applications. This is difficult and time consuming, because asynchronous events can be interleaved in arbitrary order leading to many different scenarios in the application behavior.

Inversion of Control is another, more general problem of the event-driven programming paradigm [9]. Because the asynchronous API is responsible for calling the event handler or callback function, the control over the execution of program logic is basically given to the asynchronous API (inverted). A malicious API can call the event handler or callback function at the incorrect time or multiple times resulting in unexpected application behavior.

4.2 Reo as possible solution for JavaScript’s concurrency problems

In the previous section, we saw that programming synchronization/communication in JavaScript is complex. One approach to improve this is to separate programming of computation code from programming of synchronization/communication code. For the former, programmers can perfectly use JavaScript, but for the latter (which is hard to do in JavaScript), programmers can use a special domain-specific language (DSL), specifically designed to make programming of protocols simpler. Reo is such a domain-specific language (DSL) for writing protocols.

In this thesis we will use Reo as a ‘glue language’ for the orchestration of asynchronous Web Workers and (other) asynchronous browser APIs in an effort to find a solution to the problems related to concurrent- and asynchronous programming in JavaScript highlighted
in Section 4.1. The visual representation of a Reo circuit should give better insight in the data flow of a JavaScript application and improve program comprehension.

Exogenous coordination (“coordination from outside” [2]) leads to reusable, pure coordination code. This enables reusing a single interaction protocol in multiple JavaScript applications, because the code defining the dataflow (the protocol) is not longer intertwined with application-specific logic (the processes).

Programmers can focus on writing processes (computation code) and interact with the protocol by using the API for ports. Reo ensures simple Web Workers utilization because of the simplicity of the API for ports.

Reo solves the inversion of control issues, because the resulting protocol can prohibit event-handlers or callback functions to execute when the state of the protocol does not allow this. The formal basis of Reo guarantees possibilities for formal verification, such as model checking [1], which should improve the testability of interaction- and synchronization logic.

The next two subsection give some examples of how a Reo circuit would look like and behave when the components are linked to a Web Worker of an asynchronous JavaScript API.

4.2.1 Using Reo to facilitate communication with Web Workers

Figure 4.1 shows how a Reo circuit describing the protocol of the Web Workers example from Section 2.3 looks like:

![Figure 4.1: Reo circuit describing the communication with Web Workers](image)

The main UI thread performs a put operation on the output port ‘calculateSquaredDoubled’ with a data item (e.g. ‘4’). The Reo circuit transfers this data item to the input port of the ‘Square’ Web Worker. When this worker is done processing, it puts the result on its output port and the circuit subsequently transfers this data item (‘$4^2 = 16$’) to the input port of the ‘Double’ Web Worker. When this worker is done processing, it puts the result on its output port and the circuit subsequently transfers this data item (‘$2\times16 = 32$’) to the input port ‘squaredDoubledResult’ of the main UI Thread.

4.2.2 Using Reo to orchestrate asynchronous JavaScript APIs

Figure 4.2 shows how a Reo circuit describing the protocol of the orchestration example from Section 2.2 looks like:
In the example above the JavaScript main UI thread can communicate with the Reo circuit by using its input and output ports, which are connected to a source and sink node of the circuit. The main UI thread can use the port ‘callABC’ to execute the circuit. The circuit executes asynchronous call A and B simultaneously and aggregates the result before executing asynchronous call C. Eventually, when the result of asynchronous call C is available, this result is put on the ‘resultC’ port of the main UI thread. Finally the UI thread can display this result.

4.3 Research questions

In the rest of this thesis we study the following question: “Which role can Reo play in the development of concurrent and asynchronous JavaScript programming?”

This question can be divided into a number of sub-questions:

1. “How to map Reo’s concurrency/programming model onto JavaScript’s concurrency/programming model?”

2. “Is it possible to use Reo to facilitate the communication between Web Workers and the main thread?”

3. “Is it possible to use Reo to orchestrate multiple asynchronous APIs like XHR?”

4. “Can Reo also be applied to server-side JavaScript applications?”

5. “What are the performance consequences of using Reo in asynchronous JavaScript applications?”

6. “Does Reo make it easier to understand the control flow of a concurrent JavaScript application?”

7. “How can Reo improve the testability of interaction- and synchronization logic?”

In the first four sub-questions we will study if it is at all possible to write concurrent/asynchronous JavaScript programs using Reo. In the remaining sub-questions we will study how useful Reo actually is for the development of concurrent/asynchronous JavaScript programs. The focus of this thesis is on the first four research questions. The other questions are future work.

To answer the research questions, we extended Reo with support for JavaScript as a target language. Let us use Reo’s programming workflow described in Section 3.2 as a guidance.
to describe how we implemented JavaScript support for Reo. To clarify, Figure 4.3 shows this programming workflow adapted to JavaScript terminology. The chapter numbers where the details can be found are included in this figure.

**Figure 4.3:** Programming workflow with PrDK adapted to JavaScript terminology
5. Reo runtime library for JavaScript

This chapter describes the Reo runtime library for JavaScript, which is a contribution of this thesis. Figure 5.1 shows the breakdown of the Reo runtime library for JavaScript.

![Breakdown of the Reo runtime library for JavaScript](image)

**Figure 5.1:** Breakdown of the Reo runtime library for JavaScript

The classes in the ‘runtime_core.js’ module form the core classes of Reo’s runtime library for JavaScript. Section 5.1 describes these classes and their coherence.

The Java version of Reo’s runtime uses (the flagging system of) binary semaphores to ensure the I/O operations of I/O ports complete in the correct order. In other words: These semaphores control the flow of the generated protocol. The most interesting class in the ‘runtime_core.js’ module is the ‘Semaphore’ class. This class mimics the behaviour of semaphores in JavaScript, since they are not natively supported. The ‘Semaphore’ class is described in Section 5.2.

In JavaScript, processes can either run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol). Section 5.3 describes why we distinguish between these types of processes.

Section 5.4 gives a global overview of the API for ports. Each type of process requires a different implementation of (the interface of the) API for ports. Section 5.5 describes the I/O ports for processes running in the main thread (in the context of the protocol). Section 5.6 describes the I/O ports for processes running in a background thread (in the context of a Web Worker).

To conclude, Section 5.7 describes how the individual JavaScript classes/modules are built into three bundled JavaScript modules: ‘runtime_core.js’, ‘runtime_api.js’ and ‘webworker_api.js’.
5.1 The core classes of Reo’s runtime library

The JavaScript runtime for Reo is implemented with ECMAScript 6 (as described in Section 2.4). Using the new ES6 features made it more easy to adopt the concept and structure of the Java version of the Reo runtime into the JavaScript version. JavaScript classes enabled us to transform an important part of the Reo runtime core classes in Java to their counterparts in JavaScript easily. The Reo runtime for JavaScript contains roughly the same classes as the Java version. Because of this, we will make the comparison between the Java- and JavaScript version several times to clarify the similarities and point out the difference.

This section shortly describes the interface and relation between the classes in the ‘runtime_core.js’ module. Figure 5.2 shows a global overview.

![Figure 5.2: Global overview of the classes in the ‘runtime_core.js’ module](image)

A ‘Protocol’ contains one or more ‘Automaton’ instances. Each ‘Automaton’ consists of one or more ‘State’ instances. A ‘State’ can have multiple ‘Transition’ instances. A ‘Transition’ moves an ‘Automaton’ from one ‘State’ into another (the target ‘State’). A ‘Transition’ is fired by a ‘Handler’ of a ‘Port’. The ‘Port’ instances are invoked by the processes (in case of a ‘PublicPort’ instance) or internally by the protocol (in case of a ‘PrivatePort’ instance).

The majority of the runtime core classes are abstract. Describing (the behaviour of) these classes is easier by using a generated JavaScript protocol as example. A generated JavaScript protocol will contain the concrete classes implementing the abstract classes of the runtime library. Section 8.3 and Section 8.3.1 describe (the runtime behaviour of) these classes in more details by using the ‘LateAsyncMerger2’ protocol as example.

5.2 Mimicking (the flagging system of) Semaphores in JavaScript

The Java runtime implementation of Reo uses (the flagging system of) binary semaphores to control the flow of an application. When the handler of an I/O operation cannot
complete because of invalid constraints, the I/O port blocks on a binary semaphore and waits until another I/O port signals that the constraints are changed (by releasing the semaphore).

Because of the single-threaded nature of JavaScript, blocking the execution thread (with a semaphore) will freeze the entire application. Only one function can execute simultaneously and (as a result) semaphores do not exist in JavaScript. Instead of a traditional semaphore, we need to be able to wait for the signal that the constraints are changed without blocking the execution function/thread. We decided to use the power of the Event Loop model as an alternative to implement our own ‘Semaphore’ class for JavaScript to schedule a retry for a failed handler.

```javascript
class Semaphore {
    constructor(permits) {
        this.permits = permits;
        this.queue = [];
    }
    acquire(func) {
        if (this.permits === 0) {
            this.queue.push(func);
        } else {
            this.execute(func);
        }
    }
    drainPermits() {
        this.permits = 0;
    }
    release() {
        this.permits++;
        var func = this.queue.shift();
        if (func) {
            this.execute(func);
        }
    }
    execute(func) {
        setTimeout(() => {
            func();
            this.permits--;
        }, 0);
    }
}
```

Listing 5.1: Mimicking ‘Semaphore’ class

```html
<body>

<button id="btnFoo">Click me Foo</button>
<button id="btnBar">Click me Bar</button>

</body>
</html>
```

Listing 5.2: Usage of ‘Semaphore’ class in a HTML/JavaScript application

Listing 5.1 gives a simplified implementation of this ‘Semaphore’ class which uses the power of the Event Loop model. We will create a ‘Semaphore’ instance with the given number of permits. The ‘acquire’ method accepts a function as an argument. This function will be scheduled to be executed by the Event Loop as soon there is a permit available. We can release a permit by calling the ‘release’ method.

Listing 5.2 gives an (imaginary) example of a HTML/JavaScript application using this ‘Semaphore’ class. The application has two buttons (‘btnFoo’ and ‘btnBar’) and shows the alert-box ‘FooBar’ after both buttons have been clicked (regardless of the order). The ‘click’ event-handler of button ‘btnFoo’ uses a semaphore to wait until the ‘click’ event-handler of button ‘btnBar’ releases the semaphore. The semaphore is initialized with value zero (no permits available) to ensure the alert-box is not shown until the semaphore is released.

Mimicking semaphores in JavaScript made it a lot easier to adopt the synchronization
logic of the Java version of Reo’s runtime into the JavaScript version. The main difference is that the JavaScript variant only uses the flagging system of a semaphore, while the Java variant uses semaphores also for mutual exclusion (which is not an issue in a single threaded environment like JavaScript). Section 5.5 describes how these semaphores are applied while handling I/O operations asynchronously.

5.3 Processes in background thread vs processes in main thread

A process can either run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol). Because Web Workers are relatively heavyweight [32], sometimes it is undesirable to run a process in the context of a Web Worker (in a background thread). For example, when a process only makes calls to asynchronous (built in browser) APIs which are already running in background threads (like XHR, GeoLocation, IndexDB, etc.). Instantiating a Web Worker for such scenarios will only cost extra resources and produce unnecessary overhead. As as solution, processes can also run in the main thread (in the context of the protocol). As a rule of thumb, we should execute all CPU consuming processes in a background thread (within the context of a Web Worker) and processes only orchestrating asynchronous JavaScript APIs in the main thread (within the context of the protocol).

Processes running in the context of a Web Worker correspond to the way it works in Java where each process runs in a background thread. Processes running in the main thread are currently not supported by the Java version of Reo’s runtime. This is mainly because there was no need. Threads in Java are relatively lightweight and the number of asynchronous Java APIs (using Java’s ‘Future’ interface) is limited.

5.4 The API for ports

The API for ports is responsible for ‘gluing’ JavaScript processes to a generated JavaScript protocol. The API for ports has a very simple interface: An ‘InputPort’ with a ‘get’ function and an ‘OutputPort’ with a ‘put’ function.

5.4.1 Non-blocking and asynchronous I/O operations

An important difference between JavaScript and Java is that in Java the Reo runtime ensures blocking for all I/O operations. Consequently, we do not rely on the caller to wait for the completion of I/O. Ideally, we have this in JavaScript too, but this is impossible because of the single-threaded nature of JavaScript (as described in Section 2.1). Instead, the Reo runtime for JavaScript uses (non-blocking/asynchronous) promises (as described in Section 2.2.1) for all (I/O) operations.

In contradiction to the JavaScript version, the Java version of the ‘get’/‘put’ function blocks until the I/O operations succeeds. Because of the asynchronous nature of JavaScript, any blocking operation will freeze the entire application (as described in Section 2.1). To avoid this, the JavaScript version returns a promise that will eventually resolve (with the obtained value in case of an input port) when the I/O operation succeeds. Listings 5.3 and 5.4 illustrate how an input port can be called from either Java and JavaScript.
5.4.2 Two different (and complementary) implementations

As described in Section 5.3, processes in JavaScript can either run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol). Each type of process requires a different implementation of the API for ports. Processes running in the main thread use I/O ports from the `runtime_api.js` module to communicate with the protocol. The processes and the protocol run actually in the same executing thread (the main thread) in that case. Section 5.3 describes these kind of I/O ports. Processes running in a background thread (in the context of a Web Worker) use I/O ports from the `webworker_api.js` and utilize message-passing to communicate with their counterpart on the protocol side (I/O ports from the `runtime_api.js`). The processes and protocol run completely independent in that case and do not have any shared memory. Section 5.6 describes these kind of I/O ports.

Figure 5.3 shows a global overview of the API for ports. The I/O ports in the `webworker_api.js` module are basically a layer on top of the I/O ports in the `runtime_api.js` module. When there is an I/O port on the Web Worker side (from `webworker_api.js`) there is always a counterpart I/O port on the protocol side (from `runtime_api.js`). The I/O ports on the Web Worker side utilize message-passing (in request/response style) to communicate with the `WebWorkerClient` on the protocol side as shown in Figure 5.3. The `WebWorkerClient` can be seen as a (proxy) process running in the main thread (in the context of the protocol) and routes the I/O operation from the I/O ports on the Web Worker side to their counterpart on the protocol side.

The Reo-to-JavaScript compiler is responsible for linking the applicable type of I/O port.
(from either ‘runtime_api.js’ or ‘webworker_api.js’) to the process, so programmers implementing the processes do not have to worry about this.

Section 5.5 and 5.6 respectively describe the implementation from the I/O ports of the ‘runtime_api.js’ and ‘webworkers_api.js’ module in depth. Both sections use the ‘get’ operation of an ‘InputPort’ as example. Because the I/O ports of the ‘webworkers_api.js’ depend upon the ports in ‘runtime_api.js’ (as shown in Figure 5.3) we start with the description of the latter.

5.5 I/O Ports for processes running in the main thread

This section describes the implementation of non-blocking and asynchronous I/O ports of the ‘runtime_api.js’ module in depth. A port will call its handler when an I/O operation is performed on a particular port. When the handler succeeds, both the Java and JavaScript implementation work similar and the I/O operation will complete immediately. However, when the handler fails (because of an invalid synchronization constraint) the JavaScript implementation differs from the Java implementation described in Section 3.4.

In Java, the handler is called in an infinite loop until the handler succeeds. When the handler fails, the Reo runtime ‘waits’ for another I/O operation to succeed using a binary semaphore. As soon another I/O operation succeeds, the semaphore is released and the handler-function is called again within the infinite loop earlier mentioned.

Because of the single-threaded nature of JavaScript, any infinite loop will freeze the entire application. Only one function can execute simultaneously and as a result we could not call the handler in an infinite loop. As described in Section 5.2, we decided to implement our own ‘Semaphore’ class, because we needed an approach to schedule a retry for a failed handler function without blocking the executing function/thread. The ‘acquire’ method of this ‘Semaphore’ class queues a failed handler function for retry while the ‘release’ method (called when another I/O operation succeeds) triggers the retry for this handler function.

Listings 5.5 and 5.6 give an example of respectively the Java- and JavaScript variant of the ‘tryCallHandler’ function.

An important difference between the Java- and JavaScript version of the ‘tryCallHandler’ function is that the Java function blocks until the handler succeeds and the JavaScript version returns a promise which will eventually resolve as soon as the handler succeeds. This is necessary because JavaScript should never block to avoid a freezing application. The
downside of this programming model is that we have to implement this asynchronous behavior all the way down. This means that when a method calls an asynchronous method, this method must become asynchronous too. Because the ‘tryCallHandler’ function is called from the ‘get’ operation of an input-port, this function must become asynchronous too. Listings 5.7 and 5.8 give an example of respectively the Java- and JavaScript variant of the ‘get’ function.

```java
public Object get() {
    this.buffer = null;
    this.status = IO.PENDING;
    this.handler.flag();
    this.tryCallHandler();
    this.status = IO.COMPLETED;
    return this.buffer;
}
```

Listing 5.7: Blocking ‘get’ operation of an ‘InputPort’ in Java

```javascript
get() {
    var tryGet = (resolve, reject) => {
        if (this.status === IO.PENDING) {
            reject();
        }
        else {
            this.buffer = null;
            this.status = IO.PENDING;
            this.handler.flag();
            this.tryCallHandler().then(() => {
                resolve(this.buffer);
                this.status = IO.COMPLETED;
            });
        }
    }
    return new Promise(tryGet);
}
```

Listing 5.8: Non-blocking ‘get’ operation of an ‘InputPort’ in JavaScript

Because the JavaScript version of the ‘get’ function returns a promise (immediately), an end-user can call the same port multiple times in a row. To avoid misbehavior the promise is rejected when another I/O operation is pending on the concerning port. The handler is only called when no other I/O operation is pending and the port is available. As soon as the handler function resolves, the ‘get’ function will also resolve with the retrieved value.

### 5.6 I/O Ports for processes running in a background thread

In the Java implementation of the Reo runtime, protocols and processes actually share the same instance of the input- and output ports. This is possible, because Java threads support shared memory. A major difference between threads in Java and Web Workers in JavaScript is that Web Workers cannot share memory and only support communication via message passing. This means the protocols and processes cannot share the same instance of the I/O ports. As an alternative the Reo runtime for JavaScript uses message passing (in request-response style) to facilitate the communication between protocols and processes. This communication logic is completely hidden to the end-user, giving the end-user the illusion of still using input- and output ports with shared memory.

Processes running within the context of a Web Worker (in a background thread) use the I/O ports from the ‘webworker_api.js’ module. Each I/O port on the process side has a counterpart on the protocol side (from the ‘runtime_api.js’ module). When the Web Worker process calls the ‘get’ operation of an input port, then behind the scenes the input port constructs a request message to send to the protocol. Each Web Worker is linked to a Web Worker Client on the protocol side. This client is responsible for handling the request message, invoking the corresponding port on the protocol side and sending a
response back. Sequence diagram 5.4 illustrates this communication flow.

![Sequence diagram](image)

**Figure 5.4:** Sequence diagram for the request-response behavior of an output port

5.6.1 I/O ports on the Web Worker side

The I/O ports running on the Web Worker side facilitate the communication with the protocol side. Listing 5.9 shows (a simplified version of) an input port on the Web Worker side.

```javascript
class InputPort {
  constructor(worker, name) {
    this.name = name;
    this.worker = worker;
    this.worker.addEventListener('message', this.handleResponse);
  }

  handleResponse(event) {
    var response = event.data;
    if (response.requestId === this.request.requestId) {
      this.callback(response.datum);
    }
  }

  get() {
    var tryExecuteRequest = (callback) => {
      this.request = {
        portName: this.name,
        requestId: this.get,
        method: 'get'
      };
      this.callback = callback;
      this.worker.postMessage(this.request);
    }
    return new Promise(tryExecuteRequest);
  }
}
```

**Listing 5.9:** Implementation of an ‘InputPort’ running on the Web Worker side

The ‘get’ operation wraps the I/O operation in a request object and sends this request to the protocol where it is handled. The ‘get’ function itself returns a promise which will
be resolved when the protocol sends a response back. The response from the protocol is handled by the `handleResponse` function. This function is responsible for resolving the promise with the retrieved data (by calling the temporarily stored `callback` function).

### 5.6.2 Web Worker Client on the Protocol side

Each process running in the context of a Web Worker (in a background thread) has a counterpart running in the context of the protocol (in the main thread) called the Web Worker Client. This Web Worker Client is responsible for instantiating the actual Web Worker and the communication between the Web Worker and the protocol. Listing 5.10 shows (a simplified version of) the implementation the `WebWorkerClient` class. The method `handleRequest` handles the requests sent by the process to the protocol. The request is transferred to the corresponding I/O port on the protocol side where it is handled as described in Section 5.5. The protocol sends a response (containing the retrieved data) back to the process side as soon as the I/O operation completes.

```javascript
class WebWorkerClient {
  constructor(scriptName, ...ports) {
    this.ports = ports;
    this.worker = new Worker(scriptName);
    this.worker.addEventListener('message', this.handleRequest);
  }

  handleRequest(event) {
    var request = event.data,
        port = this.getPort(request.portName);
    var promise = null;
    if (request.method === 'put') {
      promise = port.put(request.datum);
    } else if (request.method === 'get') {
      promise = port.get();
    }
    promise.then((data) => {
      var response = {
        requestId: request.requestId,
        datum: data
      };
      this.worker.postMessage(response);
    });
  }
}
```

**Listing 5.10: Implementation of the `WebWorkerClient` running on the Protocol side**

### 5.7 Build/bundle JavaScript runtime for Reo using Gulp and RollUp

Modules are the only ES6 feature used by the Reo runtime for JavaScript that are not supported by modern browsers (as described in Section 2.4.2). Getting everything working in current browsers requires a bundle step. We decided to use Rollup [37], because it is built for ES6 and supports the UMD (Universal Module Definition) pattern [38]. UMD modules are capable of working everywhere, be it on the client (using a browser), on the server (using Node.js) or elsewhere.

Gulp is a toolkit that helps to automate tasks in a JavaScript development workflow. Gulp allows us to input our source file(s), pipe them through a bunch of plugins and get an
output at the end. Gulp can be used for a lot of tasks like:

- Static code analysis (e.g. using JSHint)
- Transpiling ECMAScript 6 to ECMAScript 5 (e.g. using Babel)
- Concatenation of JavaScript files to a single file (using RollUp)
- Minification of JavaScript files

![Gulp build process for Reo's JavaScript runtime](image)

**Figure 5.5:** Gulp build process for Reo’s JavaScript runtime

The diagram of Figure 5.5 illustrates the Gulp build process. Currently, the only Gulp task for the JavaScript runtime for Reo is the concatenation/bundling of JavaScript module files into a bundled JavaScript module using RollUp. Currently three bundled JavaScript modules are created: ‘runtime.js’, ‘runtime_api.js’ and ‘webworker_api.js’.

We decided not to use a transpiler to convert the Reo runtime to ES5-compatible code supported by all browsers, because the Reo runtime is currently only using features supported by most modern browsers [33]. However, we can easily add this step (with Babel) when this becomes important in the future (e.g. because of Internet Explorer support). Also static code analysis using JSHint or minification of the resulting JavaScript module can be added to the build process easily.
6. Implementing processes

This chapter is about implementing processes. Section 6.1 describes how a programmer can implement processes in JavaScript by using the API for ports. Section 6.3 describes how to make the asynchronous process code of these processes look synchronous. A component of a Reo circuit can also be linked to a callback-based or promise-based method of an asynchronous JavaScript API. Section 6.4 describes how such an asynchronous API method looks.

6.1 Implement processes using the API for ports

Processes in JavaScript can be implemented easily by using the API for ports: We use the ‘put’ operation of an ‘OutputPort’ for producers and the ‘get’ operation of an ‘InputPort’ for consumers.

Processes using the API for ports can be implemented as a regular JavaScript function or as a JavaScript class (see Section 2.4.1 for more information about JavaScript classes). When the process is implemented as a JavaScript function, this function should accept the I/O ports and optional additional properties as argument(s). When the process is implemented as a JavaScript class, the constructor of this class should accept these argument(s) instead.

As described in Section 5.3, processes can run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol). Section 7.1 describes how to specify in which thread a process should run while linking a process to a Reo circuit. The programmer does not have to worry about this while implementing the process, because the interface of the API for ports is always the same.

6.1.1 Implement process as JavaScript function

Listing 6.1 shows some hand-written processes implemented as JavaScript function for the producer/consumer example. The functionality of this example is identical to the Java version of Listing 6.1.

The ‘get’/‘put’ operation of an ‘InputPort’/‘OutputPort’ returns a promise which will resolve with the retrieved/sent value when the I/O operations completes.

```javascript
class Processes {
    static Producer(outputPort, id) {
        String message = id + " : Hello, World!";
        outputPort.put(message).then(() => {
            Processes.Producer(outputPort, id);
        });
    }
    static Consumer(inputPort) {
        inputPort.get().then((message) => {
            console.log(message);
            Processes.Consumer(inputPort);
        });
    }
}
```

Listing 6.1: Hand-written processes in JavaScript implemented as function

The process ‘Producer’ puts a message to the output port ‘outputPort’ using the ‘put’ operation of this port. This ‘put’ method returns a promise which will resolve with the sent value when the I/O operation completes. Subsequently, the process function ‘Producer’ is called again to simulate the infinite loop of the Java variant.
The process ‘Consumer’ works similar by getting a message from the input port ‘inputPort’ using the ‘get’ operation of this port. This ‘get’ method returns a promise which will resolve with the retrieved value when the I/O operation completes. Subsequently, the process function ‘Consumer’ is called again to simulate the infinite loop of the Java variant.

### 6.1.2 Implement process as JavaScript class

Listing 6.2 shows a hand-written process implemented as an ES6 JavaScript class. The I/O-ports are given as argument to the constructor of the class. The logic of the process must be implemented in a function called ‘run’. The ‘get’ function of the ‘InputPort’ returns a promise. This promise resolves as soon as the I/O operation of the ‘InputPort’ completes and returns a value. Subsequently, the process function (‘run’) is called again to simulate the infinite loop of the Java variant.

```javascript
class Consumer {
  constructor(inputPort) {
    this.inputPort = inputPort;
  }
  run() {
    this.inputPort.get().then((message) => {
      console.log(message);
      this.run();
    });
  }
}
```

Listing 6.2: Hand-written processes in JavaScript implemented as ES6 class

### 6.2 Why blocking I/O operations are impossible in JavaScript

We still require a callback function when using an I/O port in JavaScript while one of the goals of this research was to eliminate callbacks and make asynchronous programming in JavaScript more easy. Unfortunately it is not possible to eliminate the callbacks and offer blocking I/O operations using native JavaScript. This section explains why blocking I/O operations are not possible in JavaScript.

A common remark that we have had is that blocking the execution thread of a Web Worker should not be a problem. In theory a Web Worker can just wait (block) until an I/O operation is completed without affecting the responsiveness of the application. This is indeed correct, but there is one big problem with this approach: The blocked process itself will never continue.

To illustrate that blocking I/O operations are impossible in JavaScript, consider the following Web Worker example. Remember that Web Workers can only communicate through message passing and do not share memory. The Web Worker of Listing 6.3 should send a request to the main thread and continue processing as soon as a response is received from the main thread. The event-handler of the ‘onmessage’ event sets the flag ‘responseReceived’ to true when the master sends a message back to the worker. The worker script calls the method ‘sendAndReceive’. This method sends a request to the main thread and then blocks until the response is received by polling the ‘responseReceived’ flag. However this loop is infinite, because the executing thread will never be available to handle the ‘onmessage’ event and toggle the ‘responseReceived’ flag. The implementation of the main thread and the expected (but undesired) outcome is given in Listing 6.3.
To solve the issue related to the infinite loop we have to transform the method `sendAndReceive` to an asynchronous method. An example of the changed implementation is listed in Listing 6.5. The worker script passes a callback function to the `sendAndReceive` method. A reference to this callback method is stored temporarily in the `responseReceivedCallback` variable (as a replacement for the `responseReceived` flag). The method `sendAndReceive` does not block anymore allowing the executing thread to handle the `onmessage` event and call the callback method stored in the `responseReceivedCallback` variable. The implementation of the main thread and the expected (and correct) outcome is given in Listing 6.6.

6.3 Synchronous looking process code using the API for ports

As described in Section 6.2, blocking I/O operations are impossible in JavaScript and developers still have to deal with asynchronicity when using Reo’s I/O ports in JavaScript. This section describes some code examples how to call the I/O ports in a fashion that looks more like synchronous code. The key to writing good asynchronous code is to make it look synchronous. As developers, we would much rather write processes with synchronous code like this:
function MyProcess(inputPort1, inputPort2, outputPort1) {
    var value1, value2, value3;
    value1 = inputPort1.get();
    value2 = inputPort2.get();
    value3 = value1 * value2;
    outputPort1.put(value3);
    console.log('successfully putted value 3: ' + value3);
}

Listing 6.7: Unsupported blocking I/O operations using JavaScript

6.3.1 Promises

Unfortunately, blocking I/O operations are impossible using JavaScript without freezing
the entire application. As described in Section 5.4, all I/O operations of the JavaScript
runtime for Reo return a promise. Promises are already making synchronization much
easier compared to classic callback functions (as described in Section 2.2). However when
want to implement the previous example, code complexity will still rise.

function MyProcess(inputPort1, inputPort2, outputPort1) {
    var value1, value2, value3;
    inputPort1.get().then((value) => {
        value1 = value;
        return inputPort2.get();
    }).then((value) => {
        value2 = value;
        value3 = value1 * value2;
        return outputPort1.put(value3);
    }).then() => {
        console.log('successfully putted value 3: ' + value3);
    }
}

Listing 6.8: Call non-blocking input port in JavaScript using promises

6.3.2 Async functions

With async functions (arriving in a future ECMAScript version and described in Section
2.2.2), it is possible to await on a promise. This halts the function in a non-blocking way,
wants for the promise to resolve and returns the value. Async functions will make the
previous example a lot more easy to read.

async function MyProcess(inputPort1, inputPort2, outputPort1) {
    var value1, value2, value3;
    value1 = await inputPort1.get();
    value2 = await inputPort2.get();
    value3 = value1 * value2;
    await outputPort1.put(value3);
    console.log('successfully putted value 3: ' + value3);
}

Listing 6.9: Using async function to await on non-blocking input port in JavaScript
Unfortunately async functions are not supported by modern browsers and currently only available using a transpiler like Babel.

### 6.3.3 Generator functions

Instead of async functions, we can use ES6 generator functions (described in Section 2.4.3 and denoted with ‘function*’) to create something akin to async functions. ES6 generators are supported by most modern browser in contrast to async functions.

```javascript
function* MyProcess(inputPort1, inputPort2, outputPort1) {
    var value1, value2, value3;
    value1 = yield inputPort1.get();
    value2 = yield inputPort2.get();
    value3 = value1 * value2;
    yield outputPort1.put(value3);
    console.log('successfully putted value 3: ' + value3);
}
```

Listing 6.10: Using generator function to await on non-blocking input port in JavaScript

The Reo runtime for JavaScript will automatically detect whether or not a JavaScript function linked to a component is a generator function and act accordingly. The Reo runtime for JavaScript uses a small bit of library code (a ‘spawn’ function inspired on the ‘Q’ library\(^1\)) to execute a generator function. With this ‘spawn’ function, the Reo runtime can use generator functions similar to async functions.

```javascript
function spawn(genFunc) {
    var generator = genFunc();
    function next(arg) {
        var res = generator.next(arg);
        if (!res.done) {
            return Promise.resolve(res.value)
                .then((val) => next(val));
        }
        next();
    }
    next();
}
spawn(MyProcess)
```

Listing 6.11: Call non-blocking input port in JavaScript

In the above, the generator function ‘MyProcess’ is passed to (a simplified version of) ‘spawn’. The ‘spawn’ function recursively calls ‘.next()’ on the generator, receives the promise at the yield call, and waits for it to resolve.

### 6.3.4 JavaScript compatible alternatives

Plain (regular) JavaScript code cannot suspend and wait for an I/O operation to complete. Async functions are the way to go in native JavaScript, because they do not need any external libraries or boilerplate code. Also, async functions are very recognizable for programmers knowing other languages (like C#). Until async functions are widely supported we can use Babel to transpile to ES5 or ES6 generator functions as an alternative.

---

\(^1\)https://github.com/kriskowal/q/wiki/API-Reference
Some programming languages define a superset of JavaScript and can be compiled into plain JavaScript code. TypeScript \[39\] is probably the most known of them and supports async functions (see Section \[6.3.2\]) already for some while.

### StratisfiedJS

Another language defining a superset of JavaScript is StratisfiedJS \[40\]. StratisfiedJS adds an extra ‘\texttt{waitFor}’ and ‘\texttt{resume}’ keyword to the language, allowing programmers to code with asynchronous code in a conventional, sequential style.

```javascript
function getSync () {
  waitFor(data) {
    myInputPort.get().then((result) => {
      resume(result);
    });
  }
  return data;
}
```

```javascript
var value = getSync();
console.log(value); // Logs value obtained from input port
```

**Listing 6.12:** Blocking input port call with StratisfiedJS

The example above wraps an asynchronous I/O operation of the JavaScript runtime for Reo into a synchronous StratisfiedJS function. The disadvantage of this approach is that it is not possible to call this synchronous function (‘\texttt{getSync}’) from regular JavaScript. As a result, the entire process must be written in StratisfiedJS instead of native JavaScript. Because we wanted to limit the scope of this research to native JavaScript, we decided not to study alternative languages like StratisfiedJS in detail.

### Streamline.js

Streamline.js \[41\] is another language tool to simplify asynchronous JavaScript programming. Programmers have to replace all callbacks by an underscore (‘\texttt{\_\_}’) and write their code as if all functions were synchronous. Streamline transforms the code and takes care of the callbacks.

```javascript
var value = myInputPort.get().then(_);
console.log(value); // Logs value obtained from input port
```

**Listing 6.13:** Simulate blocking input port call with Streamline.js

Streamline.js uses the same approach as async functions (discussed Section \[6.3.2\]). Instead of the ‘\texttt{async}’ and ‘\texttt{await}’ keywords, Streamline.js uses the underscore (‘\texttt{\_\_}’) keyword. Because async functions are becoming a standard in JavaScript this solution is preferable in our opinion and we decided not to implement this option.

### 6.3.5 Reo transpiler

Inspired on languages like StratisfiedJS and Streamline.js described in Section \[6.3.4\] there is another approach to give programmers the illusion of calling Reo’s I/O ports synchronously. In this approach we add a ‘\texttt{putSync}’ function to an output port and a ‘\texttt{getSync}’ function to an input port. We also include a JavaScript transpiler into the
Reo compiler package. When a process in linked to a Reo protocol and uses one of the synchronous I/O functions (‘getSync’ or ‘putSync’), the process code is automatically transpiled into an asynchronous variant using a ES6 generator function (as described in Section 6.3.3).

```javascript
function MyProcess(inputPort1)
{
    // Get value from input port
    var value = inputPort1.getSync();

    // Log first value
    console.log(value);
}
Listing 6.14: Untranspiled function with simulation of blocking ‘getSync’ function

function* MyProcess(inputPort1)
{
    // Get value from input port
    var value = yield inputPort1.get();

    // Log first value
    console.log(value);
}
Listing 6.15: Transpiled to generator function with non-blocking ‘get’ function
```

As illustrated in the example above the functions ‘getSync’ and ‘putSync’ are not supposed to be actually called at runtime. To avoid misbehaviour both operations will always throw an exception when they are accidentally called or the transpiler did not work properly. Because of time limitations, we decided not to implement this ‘Reo transpiler’ during this research project.

### 6.4 Implement processes as an asynchronous JavaScript API method

This section gives an example of an asynchronous JavaScript API method that can be linked to a component of a Reo circuit. A method of an asynchronous JavaScript API can be either callback-based (see Section 2.1.2) or promise-based (see Section 2.2.1).

Let us consider an imaginary (asynchronous) API method ‘getUser’ to retrieve an user object from a RESTful API by its identifier. This API method uses the event-based XHR (XMLHttpRequest or ‘AJAX’) API to construct a HTTP GET request to the resource ‘http://myapi/users/〈id〉’.

Listing 6.16 gives the callback-based implementation of the API method ‘getUser’. This method accepts the identifier of the user to retrieve and a callback function. This callback function will be executed when the user is retrieved. The callback function of Listing 6.16 logs the name of the retrieved user to the console. Listing 6.17 gives the promise-based implementation of the API method ‘getUser’. This method only accepts the identifier of the user to retrieve and returns a promise. When the promise resolves, then the name of the retrieved user is logged to the console.

Listing 6.18 shows how to use the promise-based version of the API method ‘getUser’ in a hand-written process using the API for ports. The process ‘GetUserProxy’ accepts two arguments: an input port and an output port. The input port ‘inputPort’ is used to get the identifier of the user to retrieve. This identifier is passed to the API method ‘getUser’ which returns a promise. Subsequently, the retrieved user is putted into the output port ‘outputPort’. As soon this I/O operation completes, the process ‘GetUserProxy’ is restarted to be ready for the next request.
```javascript
function getUser(id, callback) {
  // Do the usual XHR stuff
  var req = new XMLHttpRequest();
  req.open('GET', 'http://myapi/users/' + id);
  req.onload = function() {
    // Invoke callback with the response text
    callback(req.responseText);
  };
  // Make the request
  req.send();
// Get user with id 1234 and log name
getUser(1234, (user) => {
  console.log(user.name);
});
}
```

Listing 6.16: Callback-based ‘getUser’ API

```javascript
function getUser(id) {
  // Return a new promise.
  return new Promise((resolve) => {
    // Do the usual XHR stuff
    var req = new XMLHttpRequest();
    req.open('GET', 'http://myapi/users/' + id);
    req.onload = function() {
      // Resolve the promise with the response text
      resolve(req.responseText);
    };
    // Make the request
    req.send();
  });
// Get user with id 1234 and log name
getUser(1234).then((user) => {
  console.log(user.name);
});
```

Listing 6.17: Promise-based ‘getUser’ API

```javascript
class Processes {
  static GetUserProxy(inputPort, outputPort) {
    inputPort.get().then((id) => {
      return getUser(id);
    }).then((user) => {
      return outputPort.put(user);
    }).then(() => {
      Processes.GetUserProxy(inputPort, outputPort);
    });
  }
}
```

Listing 6.18: Hand-written processes in JavaScript calling the ‘getUser’ API method

A process like ‘GetUserProxy’ of Listing 6.18 is a perfect example of a process that should run in the main thread (within the context of the protocol). Because the XHR (XMLHttpRequest or ‘AJAX’) API already runs in the background it has no added value to spawn another background thread (Web Worker) as described in Section 5.3. This will only decrease performance because of the high amount of resources Web Workers require. In Section 7.1.2 we describe how to link a process to run in the main thread (in the context of the protocol).

Because it is a bit cumbersome to write proxies (like ‘GetUserProxy’ of Listing 6.18) for each asynchronous API method we want to link to our Reo circuit, we can also link a method of an asynchronous JavaScript API directly to a Reo circuit. Using this approach we do not need to write processes that will only serve as a proxy calling asynchronous API methods. We just specify which API method the Reo protocol should call (‘getUser’ in this particular case) and let the Reo-to-JavaScript compiler generate this proxy automatically. In Section 7.2 we describe how to do this. Using this approach we can also easily link API-methods of external libraries (like jQuery [42]) to our Reo circuit. In Section 8.2.3 we show that the generated code of the Reo-to-JavaScript compiler is very similar to our hand-coded process of Listing 6.18.
7. Link Reo circuit to JavaScript processes

Using the Reo designer, programmers can import processes by drag/dropping Java files onto the canvas. However, it is a lot of hassle to implement this feature for each supported GPL (Java, C++, JavaScript, etc.) and programmers must have the source code of the processes. A more GPL independent approach is to add generic processes (components) from the toolbox to the canvas (just like nodes, channels, etc.) and link them to a function or class implementing this process by editing component meta-data. Subsequently, a programmer can just specify a JavaScript function/class instead of a Java function/class in the meta-data of the component. Figure 7.1 shows how this feature looks like in the Reo-designer.

![Add processes and ports to the Reo designer manually.](image)

Processes can be implemented as JavaScript class/function using the API for ports or as callback/promise-based method of an asynchronous JavaScript API. Processes implemented using the API for ports can either be linked to run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol). The next sections describe how to link the various types of processes to a component of a Reo circuit. Table 7.1 summarizes these options.

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Web Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Worker Function</td>
<td>(webworker:)function-name@file-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Web Worker Class</td>
<td>(webworker:)class-name@file-name</td>
<td>Yes</td>
</tr>
<tr>
<td>Protocol Function</td>
<td>protocol:function-name@file-name</td>
<td>No</td>
</tr>
<tr>
<td>Protocol Class</td>
<td>protocol:class-name@file-name</td>
<td>No</td>
</tr>
<tr>
<td>Promise-based API</td>
<td>promise:function-name@file-name</td>
<td>No</td>
</tr>
<tr>
<td>Callback-based API</td>
<td>callback:function-name@file-name</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of how to link the various types of processes to a component in a Reo circuit

7.1 Link processes implemented using the API for ports

This section describes how to link processes implemented using the API for ports to a component of a Reo circuit. An example of the actual implementation of such a process can be found in Section 6.1 (Listings 6.1 and 6.2).
7.1.1 Link processes to run in a background thread (Web Worker)

When a process is implemented using the API for ports, it will run in a background thread (in the context of a Web Worker) by default. The name and location of the JavaScript function/class has to be specified in the meta-data of the component using the following format:

- **Function**: `function-name@file-name` (e.g. `Processes.Consumer@Processes.js` from Listing 6.1)
- **Class**: `class-name@file-name` (e.g. `Consumer@Consumer.js` from Listing 6.2)

The Reo-to-JavaScript compiler will wrap the linked JavaScript function/class into a Web Worker script allowing the Reo runtime library for JavaScript to execute the linked JavaScript function/class in a background thread within the context of a Web Worker.

7.1.2 Link processes to run in the main thread (protocol context)

As described in Section 5.3, sometimes it undesirable to run processes in the context of a Web Worker (in a background thread). Processes implemented using the API for ports can be linked to either run in a background thread (in the context of a Web Worker) or in the main thread (in the context of the protocol) by using the ‘`webworker:`’ or ‘`protocol:`’ prefixes in the component meta-data. For example:

- **Web Worker context**: `webworker:function-name@file-name`
  The JavaScript function is executed in a background thread within the context of a Web Worker. (e.g. `webworker:Processes.Consumer@Processes.js`).

- **Protocol context**: `protocol:function-name@file-name`
  The JavaScript function is executed in the main thread within the context of the protocol. (e.g. `protocol:Processes.Consumer@Processes.js`).

The example processes described in Section 6.1 (Listings 6.1 and 6.2) can be linked to either run in a background thread (within the context of a Web Worker) or in the main thread (within the context of the protocol). When no prefix is given the function is linked to run in a background thread (within the context of a Web Worker) by default.
7.2 Link asynchronous JavaScript APIs to a Reo protocol

This section describes how to link processes implemented as a method of an asynchronous JavaScript API to a component of a Reo circuit. When an asynchronous API method is linked to a component of a Reo circuit, the API method is called when all input ports of the component can fire (have a value). The values of these input ports are used as arguments for the API method. It is possible to add properties to the component when the API call requires additional arguments. The return values of the API method are put to the output port(s) of the component as soon as the API method returns a value.

Processes implemented as a method of an asynchronous JavaScript API can either be callback-based (see Section 2.1.2) or promise-based (see Section 2.2.1). A programmer has to specify the type using the ‘callback:’ or ‘promise:’ prefixes in the component meta-data. For example:

- **Callback-based API**: `callback: function-name@file-name`
  The JavaScript function is a callback-based API method (e.g. `callback: getUser@UserAPI.js`).

- **Promise-based API**: `promise: function-name@file-name`
  The JavaScript function is a promise-based API method (e.g. `promise: getUser@UserAPI.js`).

In the next two sub-sections we demonstrate this with the asynchronous API method ‘getUser’ of Section 6.4 as an example.

### 7.2.1 Link hand-written promise-based API method

Let us use the promise-based API method ‘getUser’ located in the script ‘UserAPI.js’ as an example. The implementation of this API method is given in Listing 6.17. This method accepts a single argument indicating the identifier of the user to retrieve. The method ‘getUser’ returns a promise which will resolve with the retrieved user data. To link this API method to a Reo circuit we have to create a generic component with an input port (to pass the identifier of the user to retrieve) and an output port (to return the retrieved user data). The programmer must specify the API method in the meta-data of the component using the following format: `promise: getUser@UserAPI.js`.

### 7.2.2 Link hand-written callback-based API method

Linking a callback-based API method is a bit more complex, because callback-based APIs do not have a standardized interface as promises have. A promise-based API will always return an object containing a ‘then’ function to resolve the promise. A callback-based API requires to pass a callback function, but there are no rules for the parameter index of this callback. We decided to use the parameter index of the output port to determine the parameter index of the callback function.

Let us use the callback-based API method ‘getUser’ located in the script ‘UserAPI.js’ as an example. The implementation of this API method is given in Listing 6.16. This method accepts two arguments: the identifier of the user to retrieve and a callback function. This callback function is called when the user is successfully retrieved.

To link this API method to a Reo circuit we have to create a generic component with an input port (to pass the identifier of the user to retrieve) and an output port (to return the...
retrieved user data). In this case the parameter index of the output port is important, because this must match the parameter index of the callback function (which is the second argument). Reo generates a proxy callback function that is passed to the function. This proxy callback function is responsible for routing the returned value to the output port of the component (see Section 8.2.3).

The programmer must specify the API method in the meta-data of the component using the following format: **callback:** `getUser@UserAPI.js`.

![Figure 7.3: Link callback-based ‘getUser’ method in component meta-data.](image)

### 7.2.3 Link external asynchronous JavaScript APIs (like jQuery)

Now we know how to link our hand-written asynchronous JavaScript APIs to a component of a Reo circuit, it is important to understand that external JavaScript APIs can be linked using the same approach.

Let us use the AJAX POST-method of the jQuery library as an example. This API method can both be used as a callback-based API method or as a promise-based API method. The method accepts two arguments: The URL to which the request is sent and the data that is sent to the server with the request. The callback-based version accepts a third argument containing the callback function that is executed when the request succeeds (with the response data as argument). The promise-based version returns a promise that is resolved when the request succeeds (with the response data as argument).

```
function logResult(data) {
    console.log(data);
}

$.post("http://foobar.com/newdata", "hello world", logResult);
```

**Listing 7.1:** AJAX ‘POST’ method of jQuery library using callbacks

```
function logResult(data) {
    console.log(data);
}

$.post("http://foobar.com/newdata", "hello world").then(logResult);
```

**Listing 7.2:** AJAX ‘POST’ method of jQuery library using promises

To link the AJAX POST-method of the jQuery library to a Reo circuit we have to create a generic component with a property (containing the URL), an input port (for the request data) and an output port (for the response data). A programmer can use the ‘**callback:**’ or ‘**promise:**’ prefix in the component meta-data to specify the linked method is a callback-based API or a promise-based API.

The component ‘PostMethod’ of figure 7.4 calls the AJAX POST-method of the jQuery library when a value is available in the input port ‘requestData’. The value of the property ‘url’ (‘http://foobar.com/newdata’) is used as first argument and the value of input port ‘requestData’ (e.g. ‘hello world’) as second argument. The response is putted in the output port ‘responseData’ when the promise resolves (the request succeeds).
8. Generating code

This chapter describes the (generated code of the) Reo-to-JavaScript compiler. First, Section 8.1 gives some basics about the Reo-to-JavaScript. Section 8.2 describes the compiler generated worker client classes responsible for invoking the linked processes. Section 8.3 describes the compiler generated protocol classes and their runtime behaviour. Finally, Section 8.4 describes (how to run) the compiler generated programs.

8.1 The Reo-to-JavaScript compiler

The Reo/Pr compiler is extended with an extra button to compile a Reo circuit into a JavaScript application. The Reo-to-JavaScript compiler generates a client-side HTML application to run in a web browser ('Program.html') and a server-side Node.js application to run in a terminal ('Program.js'). All other generated JavaScript files are common (shared by both the Node.js and HTML application) and implemented using the UMD (Universal Module Definition) pattern. UMD modules are capable of working everywhere, be it on the client (using a browser), on the server (using Node.js) or elsewhere.

The Reo-to-JavaScript compiler can be triggered by pressing the ‘JavaScript’ button in the ‘Reo/Pr Compiler’ window (see figure 8.1). The steps of compilation are described in Section 3.2.

50

To generate code for an automaton for an entire Reo circuit, the Reo compiler uses ANTLR's StringTemplate technology. A template is available for each target language to transform a computed automaton to source code. Implementing a compiler for a new target language is rather simple thanks to this template approach of generating code. The template-to-fill forms the only truly JavaScript-specific aspect of the Reo compiler (along with the JavaScript run-time library to actually run the generated code, of course). A compiler generated JavaScript protocol contains roughly the same classes as the Java variant.

8.2 Compiler generated Workers invoking the linked processes

The Reo-to-JavaScript generates a worker client class (‘WorkerClient.*.js’) for each component (boundary node linked to a process) the Reo circuit contains. This worker client class is responsible for invoking the linked process. The implementation of worker clients depends on the type of process linked to the component. The next sections describe the possible implementations.
8.2.1 Processes running in a background thread (Web Worker)

When a process is implemented using the API for ports, it will run in a background thread (within the context of a Web Worker) by default (as described in section 7.1.1). The Reo-to-JavaScript compiler generates the Web Worker implementation (`WebWorker*.js`) responsible for invoking the linked JavaScript class/function and the communication with the protocol.

Listing 8.1 shows (a simplified version of) the generated Web Worker code when we link the class `Consumer` from Listing 6.2 to the component `ConsumerC` of the `LateAsyncMerger2` protocol. The script first imports the Web Workers API containing the I/O ports (`webworker_api.js`) and the JavaScript file containing the process code (`Consumer.js`). Then the I/O port (`portC`) is instantiated and used to instantiate the process class (`Consumer`) implemented by the user. Finally the process is started by calling the `run` function.

Processes in JavaScript can be implemented as a traditional JavaScript function or a ES6 generator function (as described in Section 6.3.3). The `isGenerator` function detects whether or not the linked JavaScript function is an ES6 generator function. In case of a ES6 generator function, the process is started by using the `spawn` function (which implementation is given in Listing 6.11). In case of a traditional JavaScript function, the process is simply started by calling the `run` method.

```
// Import the "Web Workers" part of the JavaScript runtime library for Reo
importScripts('/pr/webworker_api.js');
// Import the JavaScript file containing the process
importScripts('/processes/Consumer.js');

// Instantiate the input port
var portC = new InputPort(this, 'portC');

// Instantiate the process
var runnable = new Consumer(portC);

// Start the process
if (isGenerator(runnable.run)) {
    spawn(runnable.run);
} else {
    runnable.run();
}
```

Listing 8.1: Generated Web Worker (`WebWorker_ConsumerC.js`)

Each generated Web Worker has a Worker Client on the protocol side. Listing 8.2 shows the generated Worker Client for the Web Worker of Listing 8.1. The generated Worker Client instantiates the generic `WebWorkerClient` class from Reo’s runtime with the (generated) Web Worker script name (`WebWorker_Consumer.js`) and ports (`portC`). This class is responsible for instantiating the actual Web Worker and the communication between the Web Worker and the protocol as described earlier in Section 5.6.
8.2.2 Processes running in the main thread (protocol context)

Processes implemented using the API for ports can also be linked to run in the main thread (within the context of the protocol) as described in Section 7.1.2. In that case no Web Worker is generated and the Worker Client instantiates and runs the process directly without spawning a background thread.

Listing 8.3 shows (a simplified version of) the generated Worker Client when we link the class `Consumer` from Listing 6.2 to the component `ConsumerC` of the `LateAsyncMerger2` protocol and specify that this process should be executed in the main thread (within the context of the protocol).

```javascript
class WorkerClient_ConsumerC {
    constructor(portC) {
        this.portC = portC;
    }

    start() {
        // Instantiate the process
        var runnable = new Consumer(this.portC);

        // Start the process
        if (isGenerator(runnable.run)) {
            spawn(runnable.run);
        } else {
            runnable.run();
        }
    }
}
```

Listing 8.3: Generated Worker Client (`'WorkerClient_ConsumerC.js'`) without Web Worker

8.2.3 Processes running asynchronously in a JavaScript API

This section describes the generated Worker Client for processes implemented as asynchronous JavaScript API. Processes implemented as asynchronous JavaScript API can either be callback-based or promise-based and the generated Worker Client will look slightly different for each case.

Listing 8.4 shows (a simplified version of) the generated Worker Client when we link the promise-based version of the asynchronous API method `getUser` (of Listing 6.17) to a component in a Reo circuit as described in Section 7.2. The API method `getUser` is called when all input ports (only `pIn` in this particular case) have fired (have a value). The value of this input port (containing the user id) is passed as argument to the API.

```javascript
class WorkerClient_ConsumerC {
    constructor(portC) {
        this.portC = portC;
    }

    start() {
        this.webWorkerClient = new WebWorkerClient('WebWorker_ConsumerC.js', this.portC);
    }
}
```

Listing 8.2: Generated Worker Client (`'WorkerClient_ConsumerC.js'`) for Web Worker
method ‘getUser’ returning a promise. As soon this promise resolves, the return value of the API method is put to the output ports (only ‘pOut’ in this particular case). When all output ports have accepted the value, the process is started again.

```javascript
class WorkerClient_GetUser {
  constructor(pIn, pOut) {
    this.pIn = pIn;
    this.pOut = pOut;
  }
  start() {
    Promise.all([this.pIn.get()]).then((args) => {
      return getUser(...args);
    }).then((result) => {
      return Promise.all([this.pOut.put(result)]);
    }).then(() => {
      this.start();
    });
  }
}
```

Listing 8.4: Generated Worker Client calling the promise-based ‘getUser’ method

Listing 8.5 shows (a simplified version of) the generated Worker Client when we link the callback-based version of the asynchronous API method ‘getUser’ (of Listing 6.16) to a component in a Reo circuit as described in Section 7.2. The only difference is that the callback-based API method ‘getUser’ is wrapped to a promise-based version (‘getUserPromise’) first. The Reo-to-JavaScript compiler assumes the callback method is the last parameter. Reo generates a proxy callback function that is passed to the ‘getUser’ function. This proxy callback function is responsible for resolving the promise.

```javascript
class WorkerClient_GetUser {
  constructor(pIn, pOut) {
    this.pIn = pIn;
    this.pOut = pOut;
  }
  getUserPromise(data) {
    return new Promise((resolve) => {
      var callback = (result) => resolve(result);
      getUser(data, callback);
    });
  }
  start() {
    Promise.all([this.pIn.get()]).then((args) => {
      return getUserPromise(...args);
    }).then((result) => {
      return Promise.all([this.pOut.put(result)]);
    }).then(() => {
      this.start();
    });
  }
}
```

Listing 8.5: Generated Worker Client calling the callback-based ‘getUser’ method

In the examples of Listing 8.5 and Listing 8.4 we assume the API method ‘getUser’ is available on the global scope. When Reo runs in a HTML application, the script containing the ‘getUser’ method is loaded via a ‘script’ tag in the header of the HTML application. When Reo runs in a Node.js application, the script is imported using the ‘require’ function.
8.3 Compiler generated JavaScript protocols

The class diagram in Figure 8.2 shows the (generated) code for the ‘LateAsyncMerger2’ protocol described in Section 3.1.1. The colored (blue) classes are the Reo runtime classes from the ‘runtime.js’ module listed in description of runtime classes of Section 5.1.

A (compiler generated) JavaScript protocol contains roughly the same classes as the Java variant described in Section 3.3 (and shown in the class diagram of Figure 3.7). We will not repeat the details of each generated JavaScript class in this chapter, because they are similar to the Java version. Section 8.3.1 describes the differences in runtime behaviour.

8.3.1 Runtime behaviour of the generated JavaScript protocol

The Reo runtime for JavaScript uses (non-blocking) promises (as described in Section 2.2.1) for all asynchronous (I/O) operations. This is a major difference with the Java version of the Reo runtime where all I/O operations are blocking (see Section 3.2.1). Because of this asynchronous behaviour, the behaviour of a generated JavaScript protocol differs from the Java version. This section describes the JavaScript runtime behavior of the protocol ‘LateAsyncMerger2’ as described in Section 3.1.3. As a comparison, the Java version of this protocol is described earlier in Section 3.4.

The sequence diagram of Figure 8.3 illustrates the behavior of the process ‘Producer A’. When the process ‘Producer A’ performs an I/O put operation on port A, this port will call its handler (‘HandlerForA’) and return a promise (‘tryPut’) to the process immediately. This promise will eventually resolve when the put operation succeeds. The handler checks if the automaton is in the correct state (‘State1’) and fires the transition ‘Transition1’. The transition checks the synchronization constraint (confirm port A has a pending I/O operation) and the data constraint.
When all constraints are valid, the value of the pending I/O operation on port A is transferred to the memory cell ‘MemoryCell1’ and the reach-method of the target state (‘State2’) is called allowing this state to become active. Eventually, the earlier returned ‘tryPut’ promise is resolved. When the synchronization constraint is invalid, the Reo runtime queues a retry of the handler function. This retry is dequeued as soon another I/O operation succeeds. The handler function will be retried using this approach until the handler function will succeed and the earlier returned ‘tryPut’ promise can be resolved. The sequence diagram for the process ‘Producer B’ will work analogously.

The sequence diagram of Figure 8.4 illustrates the behavior of the process ‘Consumer C’. When this process performs an I/O get operation on port C, this port will call its handler (‘HandlerForC’) and return a promise (‘tryGet’) to the process immediately. This promise will eventually resolve when the port yields a value. The handler checks if the automaton is in the correct state (‘State2’) and fires the transition ‘Transition3’. The transition checks the synchronization constraint (confirm port C has a pending I/O operation) and the data constraint.

When all constraints are valid, the value of the memory cell ‘MemoryCell1’ is transferred to the buffer of port C. Subsequently, the earlier returned ‘tryGet’ promise is resolved with this value. When the synchronization constraint is invalid, the Reo runtime queues a retry of the handler function. This retry is dequeued as soon another I/O operation succeeds. The handler function will be retried using this approach until the handler function will succeed and the earlier returned ‘tryGet’ promise can be resolved with the value of memory cell ‘MemoryCell1’.

Figure 8.3: Sequence diagram for the asynchronous behavior of ‘Producer A’
8.4 Compiler generated JavaScript programs

The Reo-to-JavaScript compiler generates a protocol class (‘Protocol_*.js’) for each protocol the Reo circuit contains. The details of (compiler generated) protocol classes are described in Section 8.3.1. The Reo-to-JavaScript also generates a worker client class (‘WorkerClient_*.js’) for each component (boundary node linked to a process) the Reo circuit contains. This worker client class is responsible for invoking the linked process. The details of compiler generated worker clients are described in Section 8.2.

The task of constructing ports and passing them both to the constructor of a protocol class and worker client class is performed in the compiler generated ‘Main’ class. This is a small difference compared to the Java variant, where this logic is placed in the main method (start-up routine) of the program. Putting this logic in a separate class made it easier to reuse these tasks in different types of JavaScript applications.

The Reo-to-JavaScript compiler generates a client-side HTML application (‘Program.html’) to run in a web browser (see Section 8.4.1) and a server-side Node.js application (‘Program.js’) to run in a terminal (see Section 8.4.2). Both applications use the main script (‘Main.js’) that can also be used to easily embed the Reo circuit in another application (as described in Section 8.4.3).
8.4.1 Run generated HTML application in a web browser

The generated client-side HTML application (‘Program.html’) can run in a web browser easily by opening the generated HTML file in a web browser. The generated application shows a box for each public I/O port (the boundary nodes of the circuit). With the input fields and buttons inside these boxes we can test the I/O operations of the corresponding ports. Each box also shows a log of the most recent I/O operations. Figure 8.6 shows the test page for the ‘LateAsyncMerger2’ protocol as described in Section 3.1.3.

When processes are linked to the protocol, the log will also show the I/O operations of the public ports generated by the linked processes. The log will act as a listener in that case.

8.4.2 Run generated Node.js application in a terminal

The generated ‘Program.js’ file can be executed by Node.js by invoking the command ‘node --harmony Program.js’ in a terminal. When no processes are linked to the protocol, we can communicate with the public I/O ports of the protocol by typing ‘get <<portName>>’ or ‘put <<portName>> <<value>>’ commands. Similar to the HTML
application, the Node.js application will log all I/O operations to the standard output as shown in figure 8.7.

Figure 8.7: Generated Node.js application for the ‘LateAsyncMerger2’ protocol

8.4.3 Embed a Reo circuit in another JavaScript application

The generated test applications are great for testing purposes. However, in real world scenarios we want to embed our Reo circuit (the generated protocol) in another application. Luckily we can embed the generated code in another application easily.

To embed a Reo circuit in a HTML application, we simply include the Reo runtime for JavaScript (‘runtime.js’), The API for ports (‘runtime_api.js’) and all generated JavaScript files into our application. We can start the Reo protocol by instantiating the ‘Main’ class and calling the ‘start’ method. Listing 8.6 shows how to embed a Reo circuit in a HTML application.

Embedding a Reo circuit in a Node.js application is even more simple. We just load the Main class by using the ‘require’ function. Afterwards we can start the Reo protocol by instantiating the loaded ‘Main’ class and calling the ‘start’ method. The Node.js module system will load all dependencies automatically. Listing 8.7 shows how to embed a Reo circuit in a Node.js application.
In case not all boundary nodes of our Reo circuit are linked to a process, we need to communicate with the corresponding I/O ports from our application. This is possible, because the ‘Main’ class has a property for each public I/O port. Listing 8.8 shows how to use the properties ‘A$1’ and ‘C$1’ of the ‘Main’ class to perform I/O operations.

```javascript
var Main_LateAsyncMerger = require("./Main_LateAsyncMerger");
var main = new Main_LateAsyncMerger();
main.start();

Listing 8.7: Embed a Reo circuit in a Node.js application

// put a value
main.A$1.put("myValue").then((data) => {
    console.log("putted value " + data);
});

// get a value
main.C$1.get().then((data) => {
    console.log("getted value " + data);
});

Listing 8.8: Communicate with the public I/O ports of a Reo circuit
9. Validation

During this research we tested the JavaScript version of Reo with a set of test circuits. These tests covered the all types of Reo channels and nodes and confirmed the behaviour of the JavaScript protocol matches the behaviour of the Java protocols.

Notwithstanding testing with test circuits is important, it is far more interesting to investigate whether or not Reo can be used in an actual JavaScript application. We have selected two existing reference applications that are used to validate the results of this research: One application to test an orchestration of Web Workers and another application to test an orchestration of asynchronous browser APIs. Using real world reference applications to validate our research shows that Reo can be used to develop asynchronous and concurrent JavaScript programs.

This chapter describes the two reference applications.

9.1 Reference 1: Map/reduce problem using Web Workers

The first reference application utilizes Web Workers to calculate the result of the formula \( a^b \mod c \) using the Chinese Remainder Theorem [44]. This application launches \( n \) Web Workers (threads) and each one verifies a portion of the problem set (in this case, each Web Worker computes a part of \( a^b \mod c \)).

For example, running 3 Web Workers, using \( 2^{10} \mod 7 \) as the problem:

- Web Worker #1 will compute \( 2^4 \mod 7 \);
- Web Worker #2 will compute \( 2^3 \mod 7 \);
- Web Worker #3 will compute \( 2^3 \mod 7 \);

In the end the main thread will use all three results and compute the final one. This can be seen as a map/reduce problem. A live running example with a benchmark is available [44]. Figure 9.1 shows a screenshot of this application calculating the formula \( 2^{102400000} \mod 97777 \), with respectively one, two, three, four or five Web Workers.

![Figure 9.1: Benchmark test for the calculation of \( 2^{102400000} \mod 97777 \) using Web Workers](https://github.com/pmav/web-workers)

We downloaded the source code of this application from GitHub [1] and divided all computational code from the synchronization/communication logic. Subsequently, we copied the computational code into Reo processes and replaced the synchronization/communication by a Reo circuit.

9.1.1 Designing the protocol

The Reo circuit (shown in Figure 9.2) of the ‘map/reduce’ protocol defines five processes: A process (‘DividePayload’) to divide the problem into equal portions of sub-problems, three processes (‘Compute0..2’) computing a single sub-problem and a process (‘MergePayload’) to merge the sub-results into a final result. The process ‘DividePayload’ consumes the entire problem as input and produces the three sub-problems as output (by performing three sequential ‘put’ operations on the port ‘payloadOut’). The protocol distributes those three sub-problems to the three ‘Compute’ processes. Each ‘Compute’ process obtains his portion of the problem and calculates a sub-result. Each sub-result is consumed by the process ‘MergePayload’. This process merges each sub-result to produce a final result.

The port ‘power’ should be used to pass the value of ‘b’ (of the formula ‘a^b \mod c’). The values of ‘a’ and ‘c’ are configured as properties of the ‘DividePayload’ process (respectively ‘base’ and ‘modulo’). The port ‘result’ should be used to obtain the final result.

A problem with Web Workers is that they are allowed to send messages to the main thread at any time. This causes inversion of control as described in Section 4.1. An (optional) improvement is to disallow the Web Workers to perform I/O operations when the state of the protocol does not allow this. The processes ‘Compute0..2’ are only allowed to produce a value once (and only once) when the protocol has requested to do so. With some slight modifications we can define a protocol that ensures the correct execution sequence. The SyncDrain channel connected to the port ‘power’ in combination with the full FIFO channel ensures that initially only the port ‘power’ can fire. Subsequently, the ring of FIFO channels in combination with their connected SyncDrain ensures the ports ‘compute0..2In’ are one after the other allowed to fire. Finally, only the port ‘mergeOut’ is allowed to fire. When this I/O operation completes, the port ‘power’ can fire again.
9.1.2 Writing the processes

We will briefly describe the details of the Reo processes.

**DividePayLoad**

The ‘DividePayLoad’ process of Listing 9.1 is responsible for dividing the problem into equal portions. The argument ‘workerNumber’ defines the number of portions (equal to the number of Web Workers). The arguments ‘base’ and ‘modulo’ are used to define respectively the values of ‘a’ and ‘c’. The remaining arguments are the input- and output ports. The input port ‘pIn’ is used to obtain the power (the value of ‘b’). The function ‘UTIL.powerDivision’ is the original version from GitHub and returns an array of three items. For each item a payload object is constructed (defining a portion of the problem) and potted to the output port ‘pOut’.

```plaintext
static DividePayload(pIn, pOut, base, modulo, workersNumber) {
    var power = yield pIn.get(); // Get the power (value of b)

    // Divide the work into equal portions
    var powers = UTIL.powerDivision(power, workersNumber);
    for (var i = 0; i < powers.length; i++) {
        var payload = { base: base, power: powers[i], mod: modulo }
        yield pOut.put(payload);
    }
}
```

Listing 9.1: Implementation of the ‘GetPayload’ process

**Compute**

The ‘Compute’ process of Listing 9.2 is responsible for the actual computation of a portion of the problem. The input port ‘pIn’ is used to obtain the payload. The function
'UTIL.power' is the original version from GitHub and does the actual computation. The result is putted to the output port 'pOut'.

```java
static Compute(pIn, pOut, index) {
    var payload = yield pIn.get(); //Get the portion of the problem
    //Do the computation
    payload.power = UTIL.power(payload.base, payload.power, payload.mod);
    yield pOut.put(payload); //Put the result of the computation
}
```

Listing 9.2: Implementation of the 'Compute' process

**MergePayload**

The 'MergePayload' process of Listing 9.3 is responsible for merging the result of each computation into a final one. The argument 'workerNumber' defines the number of portions (equal to the number of Web Workers). The remaining arguments are the input- and output ports. The input port 'pIn' is used to obtain the result of each portion of the problem (payload[0..2]). As soon as all sub-results are available, the final result is putted to the output port 'pOut'.

```java
static MergePayload(pIn, pOut, workersNumber) {
    var result = 1;
    //Await the result of each computation and compute the final result
    for (var i = 0; i < workersNumber; i++) {
        let payload = yield pIn.get();
        result = (result * payload.power) % payload.mod;
    }
    yield pOut.put(result); //Put the final result
}
```

Listing 9.3: Implementation of the 'MergePayload' process

**9.1.3 Running the test application(s)**

When we run the generated client-side HTML application ('Program.html') we can easily test our implementation of the Reo circuit. We put the value '1024000000' to the port 'power$1' and subsequently get the value from the port 'result$1' to inspect the final result. The result shown in Figure 9.4 matches the result shown in the original application shown in Figure 9.1.

![Figure 9.4: Generated HTML application for the 'map/reduce' protocol](image)

It is good to know that the generated Node.js application ('Program.js') works as well (as shown in figure 9.5). We use the command 'get power$1 1024000000' to put the value
1024000000 to the port ‘power1’. Subsequently we use the command ‘get result1’ to get the final result.

Figure 9.5: Generated Node.js application for the ‘map/reduce’ protocol

9.1.4 Understanding the original reference application

Before embedding our Reo circuit into the original version of the reference application from GitHub, let us have a look at the details of this application. The application has a ‘Main’ and ‘Util’ object and a complementary Web Worker script named ‘Worker.js’.

The ‘Util’ object contains some utility functions to support the calculation (e.g. ‘powerDivision’ and ‘power’). We already briefly described these functions, because we (re)used them in the Reo processes of Listing 9.1 and Listing 9.2.

The ‘Main’ object contains three important functions: ‘run’, ‘sendMessage’ and ‘callback’. The ‘run’ function starts the calculation for a certain input and is responsible for dividing the payload into equal portions. For each portion a Web Worker is instantiated and started. The ‘run’ function can be seen as the counterpart of the Reo process ‘DividePayload’ (it uses the same ‘powerDivision’ function of the ‘Util’ class). The ‘sendMessage’ function is a helper function to send messages to the Web Workers. The event messages from the Web Workers are handled in the ‘callback’ function. This function is responsible for the aggregation of Web Worker results and can be seen as the counterpart of the Reo process ‘MergePayload’. Finally the result is displayed in the UI by using the ‘log’ method of the ‘Util’ object. Listing 9.4 shows the original implementation of the ‘Main’ class.

```javascript
MAIN = {
  workersNumber : 0,  // Total number of Web Workers (dynamic).
  workersEnded : 0,   // Number of terminated Web Workers.
  base : null,
  power : null,
  mod : null,
  startTime : null,   // Current test start time.
  results : null,     // Current test results (for debugging).
  testNumber : 1,     // Current test number.
  run : function(workersNumber, base, power, mod) {
    var i, powers;
    // Initial setup.
    this.workersEnded = 0;
```

64
this.workers = [];
this.results = [];
this.workersNumber = workersNumber;
this.base = base;
this.power = power;
this.mod = mod;

// Divide work by Workers.
powers = UTIL.powerDivision(power, workersNumber);

// Setup Workers.
for (i = 0; i < workersNumber; i++) {
  this.workers[i] = new Worker("Worker.js");
  this.workers[i].onmessage = this.callback;
  this.workers[i].onerror = this.error;
}

this.startTime = new Date();

// Start workers.
for (i = 0; i < workersNumber; i++) {
  this.sendMessage(i, base, powers[i], mod);
}

sendMessage : function(workerId, base, power, mod) {
  var data = {};
  data.workerId = workerId;
  data.payload = {};
  data.payload.base = base;
  data.payload.power = power;
  data.payload.mod = mod;
  this.workers[workerId].postMessage(JSON.stringify(data));
},
callback : function(event) {
  var i, result, data = JSON.parse(event.data);
  var workerId = data.workerId;
  var payload = data.payload;
  MAIN.workers[workerId].terminate();
  MAIN.workersEnded += 1;
  MAIN.results[workerId] = payload.result;

  if (MAIN.workersEnded === MAIN.workersNumber) { // All Web Workers done.
    // Merge results.
    result = 1;
    for (i = 0; i < MAIN.results.length; i++) {
      result = (result * MAIN.results[i]) % MAIN.mod;
    }

    var time = ((new Date) - MAIN.startTime);

    // Output run results.
    var output = MAIN.testNumber + "^" + MAIN.workersNumber + " Workers, Test: "
      +MAIN.base+"^"+MAIN.power+" mod "+MAIN.mod+" = "+result+", "+ time + " ms ";

    UTIL.log(output);
  }
}

Listing 9.4: Original implementation of the ‘Main’ object

The three Web Workers instantiated by the ‘run’ function of the ‘Main’ class execute the
script ‘Worker.js’ in a background thread. The functionality of the Web Worker script
can be seen as the counterpart of the Reo process ‘Power’ (it uses the same ‘power’
function of the ‘Util’ class). The implementation of this Web Worker script is shown in
Listing 9.5

```javascript
importScript('Util.js');

self.onmessage = function(event) {
  // Message from MAIN.sendMessage()
  var data = JSON.parse(event.data);
  var base = data.payload.base;
  var power = data.payload.power;
  var mod = data.payload.mod;
  data.payload.result = UTIL.power(base, power, mod);
  postMessage(JSON.stringify(data)); // Callback to MAIN.callback().
};
```

Listing 9.5: Implementation of the worker script ‘Worker.js’.

9.1.5 Embed the Reo circuit into the reference application

Now we have completed/tested our Reo application and have clear understanding of the original version of the reference application, it is time to embed our Reo circuit into this application.

```javascript
MAIN = {
  testNumber : 1, // Current test number.
  startTime : null, // Current test start time.

  run : function(workersNumber, base, power, mod) {
    this.startTime = new Date();
    var protocol = new Main_ChineseRemainderTheorem();
    protocol.start(); // Start the Reo circuit
    protocol.power$1.put(power);
    protocol.result$1.get().then((result) => {
      var time = ((new Date) - MAIN.startTime);
      // Output run results.
      var output = MAIN.testNumber + "\nWorkers, Test: \n" + base + "^" + power + " mod " + mod + " = " + result + ", time = " + time + " ms"
      UTIL.log(output);
      MAIN.testNumber += 1;
    });
  }
}
```

Listing 9.6: Embed the ‘map/reduce’ protocol into the original application

We simplify the ‘Main’ class of the original application and plug in our Reo circuit into the ‘run’ function. Instantiating, starting and communicating (with) the Web Workers is an implementation detail of the Reo circuit. This is not the responsibility of the ‘Main’ object anymore. We simplify the ‘run’ function and completely remove the ‘callback’ and ‘sendMessage’ function. The final implementation is shown in Listing 9.6. First the Reo circuit is instantiated and started. Then the input value (power) is put into the output port ‘power$1’ and subsequently we invoke a get operation on the input port ‘result$1’ to obtain the end result. Finally, when this I/O operations completes, we display the end result in the UI. A screenshot of the web page is shown in Figure 9.6.
9.1.6 Performance measurement

We compare the performance of the original version of the reference application from GitHub with the new version using the Reo circuit. Both applications calculate the problem \(2^{1024000000} \mod 97777\) with respectively one to ten workers. For better results we run the test five times and calculated the average. The results are shown in Table 9.1.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Original version</th>
<th>Reo version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8929 ms</td>
<td>8908 ms</td>
</tr>
<tr>
<td>2</td>
<td>4571 ms</td>
<td>4614 ms</td>
</tr>
<tr>
<td>3</td>
<td>3130 ms</td>
<td>3160 ms</td>
</tr>
<tr>
<td>4</td>
<td>2393 ms</td>
<td>2462 ms</td>
</tr>
<tr>
<td>5</td>
<td>2395 ms</td>
<td>2481 ms</td>
</tr>
<tr>
<td>6</td>
<td>2427 ms</td>
<td>2468 ms</td>
</tr>
<tr>
<td>7</td>
<td>2448 ms</td>
<td>2460 ms</td>
</tr>
<tr>
<td>8</td>
<td>2702 ms</td>
<td>2500 ms</td>
</tr>
<tr>
<td>9</td>
<td>2737 ms</td>
<td>2514 ms</td>
</tr>
<tr>
<td>10</td>
<td>2623 ms</td>
<td>2555 ms</td>
</tr>
<tr>
<td>Total</td>
<td>34355 ms</td>
<td>34122 ms</td>
</tr>
</tbody>
</table>

Table 9.1: Performance measurement: original version vs Reo version

The majority of the time is probably consumed by the processes. However, we were mainly interested in the time consumed by the synchronization/communication logic. To measure this, we decided to remove the computational logic from the processes (and just immediately return a constant value). This remaining time is consumed by the synchronization/communication logic. Table 9.2 shows the results.

The performance measurement was done in Firefox on an Intel(R) Core (RM) i5-4200M CPU @2.5GHz with 8GB RAM. Other browsers showed similar results. We carefully conclude that the Reo version has similar performance than the original version of the reference application. In the variant with computational logic, the Reo version performs slightly better. In the variant without computational logic, the original version performs slightly better. We should repeat our tests with a bigger number (than five) of repetitions to remove noise from our performance measurements.
Table 9.2: Performance measurement of synchronization/communication logic

<table>
<thead>
<tr>
<th>Workers</th>
<th>Original version</th>
<th>Reo version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>2</td>
<td>18 ms</td>
<td>22 ms</td>
</tr>
<tr>
<td>3</td>
<td>20 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>4</td>
<td>24 ms</td>
<td>31 ms</td>
</tr>
<tr>
<td>5</td>
<td>29 ms</td>
<td>36 ms</td>
</tr>
<tr>
<td>6</td>
<td>33 ms</td>
<td>39 ms</td>
</tr>
<tr>
<td>7</td>
<td>39 ms</td>
<td>45 ms</td>
</tr>
<tr>
<td>8</td>
<td>41 ms</td>
<td>50 ms</td>
</tr>
<tr>
<td>9</td>
<td>44 ms</td>
<td>54 ms</td>
</tr>
<tr>
<td>10</td>
<td>51 ms</td>
<td>59 ms</td>
</tr>
<tr>
<td>Total</td>
<td>314 ms</td>
<td>380 ms</td>
</tr>
</tbody>
</table>

9.1.7 Evaluation

Now we have embedded our Reo circuit into the original version of the reference application, it is time to evaluate how successful it was.

The benefit of using the Reo circuit was that the division of responsibilities became more clear. We were able to isolate the processes into JavaScript functions with a single responsibility (respectively dividing the payload, merging the payload and calculating a portion of the payload). The protocol is (only) responsible for the interaction between these processes. This enables the possibility to reuse the protocol in another application for another map/reduce problem.

Another benefit is that the visual representation of the Reo circuit helped to understand the control-flow more easily. This is also the main reason we started with describing the application in Reo instead of the original reference application. Last but not least, with the Reo circuit we can prevent the Web Worker processes from performing I/O operations when the protocol does not allow this. The processes ‘Compute$^{0..2}$’ are only allowed to produce a value when the protocol has requested to do so. In the reference application, the Web Workers can send messages to the main thread at any given moment. This causes inversion of control as described in Section 4.1.

On the downside we have to say that we were not able to implement one of the features of the original reference application. The application had a slider to dynamically (at runtime) configure the number of Web Workers. This is not possible in Reo, because the number of workers is fixed. We tried to use Pr (Reo’s textual superset as described in Section 3.1.2) to write a protocol with a variable number of workers. Despite the fact that we were able to do this (and used in for our performance measurement), the circuit still needs a compilation step before it can be used in a JavaScript application. So, currently it is not possible to instantiate a Reo circuit with a variable number of workers and we decided to only run the application with a fixed amount of three workers.

Another disadvantage is that Reo requires the PrDK Development Kit, while most JavaScript developers are used to only need a text-editor to develop their programs. Besides this, designing Reo circuits has a certain learning curve. In that perspective, JavaScript developers might prefer other JavaScript libraries to simplify the communication with Web Workers. Some examples are like Parallel.js, Ooperative and Hamsters.js.
9.2 Reference 2: Weather using GeoLocation- and Weather API

The second reference application orchestrates the GeoLocation API and the OpenWeatherMap API to show the current weather of the current location. This application invokes the GeoLocation API to obtain the current location and subsequently invokes the OpenWeatherMap API to obtain the actual weather for this location. The functionality of this application is similar to the website [http://degreees.com/](http://degreees.com/) showing the current weather for the current location.

Unfortunately we did not have the source code of this website. As an alternative we decided to implement a similar web page our self.

![Screenshot of the website](http://degreees.com/)

**Figure 9.7:** Screenshot of the website [http://degreees.com/](http://degreees.com/)

9.2.1 Describing the processes

We did not have to write any hand written Reo process to implement this reference application. The application is just an orchestration of the GeoLocation API and the OpenWeatherMap API. This section shortly describes these APIs.

GeoLocation API

The GeoLocation API allows the user to provide their location to web applications if they so desire. For privacy reasons, the user is asked for permission to report location information.

The GeoLocation API is published through the `navigator.geolocation` object. To obtain the user’s current location, we should call the `getCurrentPosition` method. This initiates an asynchronous request to detect the user’s position, and queries the positioning hardware to get up-to-date information. When the position is determined, the defined callback function is executed. We can optionally provide a second callback function to be executed if an error occurs. A third, optional, parameter is an options object where we can set the maximum age of the position returned, the time to wait for a request, and if we want high accuracy for the position.

The example of Listing 9.7 will cause the `do_something` function to execute when the location is obtained. The second parameter (`error`) and third parameter (`options`) are explicitly specified, but can also be omitted when the value is undefined.
```javascript
var callback = function(position) {
    do_something(position.coords.latitude, position.coords.longitude);
};
var error = undefined;
var options = undefined;
navigator.geolocation.getCurrentPosition(callback, error, options);
```

Listing 9.7: Example of the usage of the ‘getCurrentPosition’ method

**OpenWeatherMap API**

The OpenWeatherMap API is simple, clear and free. To access the API we needed to sign up for an API key. The HTTP GET operation to the URL `http://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={appid}` retrieves the current weather by geographic coordinates.

We used the AJAX GET-method of the jQuery library (as described in Section 7.2.3) to execute the HTTP call. This API method accepts two arguments: The URL to which the request is sent and the data that is sent to the server with the request. In this particular case we use the URL to the OpenWeatherMap API and an object describing the geographical location and the API key.

The example of Listing 9.8 will cause the ‘do_something’ function to execute when the current weather is obtained for a specific geographical location.

```javascript
var requestData = {
    lat: 51.624452,
    lon: 4.850029,
    appid: 'foobar',
    units: 'metric'
};
var url = "http://api.openweathermap.org/data/2.5/weather";
$.get(url, requestData).then(function(result) {
    do_something(result);
});
```

Listing 9.8: AJAX ‘GET’ method of jQuery library retrieving current weather

### 9.2.2 Designing the protocol and linking the processes

The Reo circuit (shown in Figure 9.8) defines two processes: A process (‘GeoLocation’) to get the current geographical position and a process (‘Weather’) to get the current weather for a specific geographical position. The port ‘input’ should be used to start the process and the port ‘output’ should be used to obtain the final result.

The process ‘GeoLocation’ is linked to the callback-based ‘getCurrentPosition’ method of the GeoLocation API. The parameter index of the output port must match the parameter index of the callback function (which is the first parameter) as described in Section 7.2.2. The parameter index of the input port is set to four, because the method ‘getCurrentPosition’ does not need any input. Because the method ‘getCurrentPosition’ only accepts three parameters, the value of the input port is ignored. The location is put to the output port when the location is obtained.

We have defined a transform channel linked to the method ‘GetCoords’ to transform the location returned by the GeoLocation API to a data object that the OpenWeatherMap
API accepts as request.

The process ‘Weather’ uses the AJAX GET-method of the jQuery library and is linked to the promise-based ‘get’ method. The property ‘url’ defines the URL to the OpenWeatherMap API. The input port is used to pass the data that is sent to the server with the request (the geographical location the API key). The weather data is put to the output port when the weather is obtained.

We have defined a second transform channel linked to the method ‘FormatWeather’ to transform the response returned by the OpenWeatherMap API to a data object with only the specific information we want to show in our weather application.

![Figure 9.8: A Reo circuit visualizing the synchronization/communication logic](image)

9.2.3 Running the test application

When we run the generated client-side HTML application (‘Program.html’) we can easily test our implementation. We put a value to the port ‘input$1’ and subsequently get the value from the port ‘output$1’ to inspect the final result. Luckily the result shown in Figure 9.9 matches the result shown in the original application shown in Figure 9.7.

![Figure 9.9: Generated HTML application for the ‘CurrentWeather’ protocol](image)

Because the GeoLocation API is not available outside of the browser we could not run this circuit in Node.js.

9.2.4 Embed the Reo circuit into a web page

Listing 9.9 shows a simple web page we created to show the current weather obtained from our Reo circuit.
Everything within the ‘head’ tag is copied from the generated client-side HTML application (‘Program.html’). Only the HTML within the ‘body’ tag and the inline JavaScript is hand written for this application. First the Reo circuit is instantiated and started. Then the port ‘input$1’ is used to request the current weather. Subsequently the port ‘output$1’ is used to obtain result. Finally, when this I/O operations completes, we show weather in the UI. A screenshot of the web page is shown in Figure 9.10.

### 9.2.5 Performance measurement

To measure the time consumed by the synchronization/communication logic we replaced the two API calls by two fakes (‘FakeGeoLocation’ and ‘FakeWeather’). Those two

---

**Listing 9.9:** Simple web page showing the weather obtained from our Reo circuit

```html
<html>
<head>
  <script src="/pr/runtime_core.js"></script>
  <script src="/pr/runtime_api.js"></script>
  <script src="/processes/jQuery.js"></script>
  <script src="/protocols/CurrentWeather.js"></script>
  <script src="/Protocol_CurrentWeather.js"></script>
  <script src="/WorkerClient_Get.js"></script>
  <script src="/WorkerClient_GetCurrentPosition.js"></script>
  <script type="text/javascript">
    var main = new Main_Weather();
    main.start();

    main.input$1.put('');
    main.output$1.get().then(showWeather);

    function showWeather(jsonData) {
      $('#weatherPlace').html(jsonData.place);
      $('#weatherTemp').html(jsonData.temp + " degrees");
      $('#weatherIcon').attr('src', "http://openweathermap.org/img/w/" + jsonData.icon + "\:.png");
    }
  </script>
</head>
<body>
  <div id="weather">
    <h1 id="weatherPlace"></h1>
    <h2>
      <img id="weatherIcon" />
      <span id="weatherTemp"></span>
    </h2>
  </div>
</body>
</html>
```

---

**Figure 9.10:** Screenshot of the weather web page using the Reo circuit
fakes return immediately, so the remaining time should be consumed by the protocol. Because we did not have any reference, we hand-coded a simple reference application doing the same. Table 9.3 shows the results.

<table>
<thead>
<tr>
<th>Run</th>
<th>Hand-coded version</th>
<th>Reo version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>2</td>
<td>14 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>3</td>
<td>11 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>4</td>
<td>9 ms</td>
<td>16 ms</td>
</tr>
<tr>
<td>5</td>
<td>12 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>Total</td>
<td>56 ms</td>
<td>75 ms</td>
</tr>
<tr>
<td>Average</td>
<td>11.2 ms</td>
<td>15 ms</td>
</tr>
</tbody>
</table>

Table 9.3: Performance measurement of synchronization/communication logic

Again, the performance measurement was done in Firefox. Unfortunately, the Reo implementation turned out to be slightly slower compared to the hand-coded version. This is probably caused by the fact that Reo wraps each API call into a process using asynchronous I/O port. The I/O calls to these ports might cause the overhead. Future research is needed to confirm this.

9.2.6 Evaluation

Now we have implemented our own weather application using Reo, it is time to evaluate how successful it was.

Using Reo to orchestrate asynchronous JavaScript APIs felt a bit like using a sledgehammers to crack nuts. Designing an entire Reo circuit for such a simple use-case seems a bit overkill. Using native JavaScript constructs like promises might be more convenient in such a case.

A benefit of using Reo is that we have created a reusable component to obtain the current weather for the current weather. We can easily reuse this component in another application. Of course, such a component can also be coded in native JavaScript. But Reo might help developers to think of reusable components (or protocols) upfront.

Implementing a more complex use-case to unveil possible other advantages is considered to be future work.
10. Conclusion

In this chapter we present our conclusion of this research project. Section 10.1 summarizes the answers to the research questions from Section 4.3. Section 10.2 evaluates which role Reo can actually play in the development of concurrent and asynchronous JavaScript programs. Section 10.3 discusses the relation of our work with the work of others. Section 10.4 suggest a number of areas for further research.

10.1 Answers to research questions

Before answering the research question “Which role can Reo play in the development of concurrent and asynchronous JavaScript programming?”, we will first briefly describe the answers of the sub-questions described in Section 4.3.

1. How to map Reo’s concurrency/programming model onto JavaScript’s concurrency/programming model?

The answer on this research question is the main contribution of this thesis and especially Chapter 5. We will give a short summary.

Multithreading: Thanks to the Web Workers API, JavaScript meets Reo’s most important requirement of supporting some form of multithreading.

Shared memory vs Message passing: A major difference between Web Workers in JavaScript and threads in other GPLs (like Java or C) is that Web Workers cannot share memory and only support communication via message passing. This means the protocol and processes cannot share the same instance of I/O ports. As an alternative the JavaScript version of the API for ports uses message passing (in request-response style as described in Section 5.6) to facilitate the communication between protocols and processes. This communication logic is completely hidden to the end-user, giving the end-user the illusion of still using I/O ports with shared memory.

Blocking I/O operations vs Non-blocking I/O operations: The main discrepancy between GPLs conforming to the definitions of Reo (like Java and C) and JavaScript are (the lack of) blocking I/O operations. A property of the Event Loop model is that JavaScript never blocks. Because of the single-threaded nature of JavaScript, blocking the execution thread will freeze the entire application (as described in Section 2.1).

In Section 5.4 we introduced non-blocking and asynchronous I/O operations (in combination with callbacks and/or promises) as an alternative. However, non-blocking I/O operations are in conflict with the definitions of Reo where all I/O operations should be blocking. We had to find a way to mimic the blocking I/O operations of Reo into the non-blocking principles of JavaScript. In Section 6.3 we described how to make asynchronous (callback- or promise-based) code look more like synchronous code.

Concurrent processes vs Asynchronous APIs: Although Web Workers work well for implementing (hand-written) background processes, they are not useful when we want to use Reo to orchestrate asynchronous JavaScript APIs. This is mainly because of two reasons: First, the JavaScript runtime will run these APIs in a thread pool (and thus already running in a background thread). Second, Web Workers are relatively heavy-weight, and are not intended to be used in large numbers. Instantiating a Web Worker for such scenario’s will only cost extra resources and
produce unnecessary overhead. It is more efficient to call asynchronous JavaScript APIs directly from the main thread.

In Section 5.3 we defined an approach to distinguish between concurrent processes (running in a background thread) and asynchronous processes (running in the main thread). This allows us to execute all CPU consuming processes in a background thread and processes only orchestrating asynchronous JavaScript APIs in the main thread.

2. Is it possible to use Reo to facilitate the communication between Web Workers and the main thread?

Processes can be implemented easily by using the API for ports (as described in Section 5.4 and Section 6.1). These processes can be linked to a Reo circuit (as described in Section 7.1.1). The Pr-to-JavaScript compiler wraps these processes into a Web Worker to ensure the processes are executed in a background thread (as described in Section 8.2).

In Section 9.1 we described a reference application utilizing Web Workers. We divided all computational code from the synchronization/communication logic. Subsequently, we copied the computational code into Reo processes (using the API for ports) and replaced the synchronization/communication by a Reo circuit. We demonstrated that the functionality of this reference application remained the same.

3. Is it possible to use Reo to orchestrate multiple asynchronous APIs like XHR?

Processes can be implemented as an asynchronous JavaScript API method (as described in Section 6.4). Section 7.2 describes how to link an asynchronous API method directly to a Reo circuit (without writing any code). The Pr-to-JavaScript compiler wraps these API methods into a Reo process responsible for invoking the linked process (as described in Section 8.2.3).

In Section 9.2 we described a reference application showing the current weather for the current geographical location. We have implemented an application with similar functionality using the HTML5 GeoLocation API and the XHR/AJAX API (calling the OpenWeatherMap API). The protocol we defined is linked directly to those asynchronous APIs, eliminating the need of writing any process code.

4. Can Reo also be applied to server-side JavaScript applications?

As described in Section 9.1.3, we were able to execute the first reference application in Node.js to show that a Reo circuit can be applied in server-side JavaScript applications as well.

The focus of this thesis was on the four research questions (as described in Section 4.3). The following questions were considered to be future work. Nevertheless, this thesis already provides some preliminary insight.

5. What are the performance consequences of using Reo in asynchronous JavaScript applications?

We execute a performance measurement for both reference application as described in Section 9.1.6 and Section 9.2.5. The results for the first reference application (using Web Workers) are promising and show that Reo version has similar performance than the original version of the reference application. The result for the second reference application (using an orchestration of asynchronous APIs) was a
bit disappointing. The performance of the hand-written protocol was better than the compiler generated Reo protocol. This is probably caused by the extra overhead of the asynchronous I/O ports.

6. Does Reo make it easier to understand the control flow of a concurrent JavaScript application?

In Chapter 9 we described some insights on this topic. In Section 9.1.7 we found out that it was easier to describe the control flow of the first reference application with our Reo circuit. On the other hand, in Section 9.2.6 we found out that the control flow should reach a certain level of complexity before using Reo circuit becomes an advantage. For simple use-cases (like our second reference application) Reo will only complicate things.

7. How can Reo improve the testability of interaction- and synchronization logic?

With Reo it is easy to isolate the interaction- and synchronization logic, because this logic (the protocol) is not longer intertwined with application-specific logic (the processes). Reo isolates processes from protocols and allows us to test protocols as a separate unit more easily. For example by using mock processes.

The formal foundation of Reo also enables formal analysis (like model checking). This should make statically verifying a given protocol relatively easy. Unfortunately, we did not have time to experiment with these possibilities in combination with JavaScript.

Before returning to the research question “Which role can Reo play in the development of concurrent and asynchronous JavaScript programming?”, we first have to emphasise that it is possible to write concurrent/asynchronous JavaScript programs using Reo. Our main concern is how useful Reo actually is in the development of real world JavaScript programs. We will describe our thoughts in the next section.

10.2 Evaluation

Using Reo, the division of responsibilities in JavaScript applications becomes more clear. Processes are nicely isolated into JavaScript functions/classes with a clear and single responsibility and protocols connecting them are defined separately. This encourages the reuse of both protocols and processes. However, we do not necessarily need Reo (or another DSL) to achieve this. We could also design a JavaScript design pattern for implementing (reusable) protocols and processes easily.

Utilizing unabstracted Web Workers is cumbersome. Reo eliminates the need to design/implement message-passing contracts and write event listeners ourselves, by using the (simple) API for ports to allow communication between Web Workers and the main thread. A well designed Reo protocol will also prevent inversion of control, by allowing processes only to perform I/O operations when the protocol has requested to do so. However, JavaScript developers might prefer native JavaScript libraries (like Parallel.js, Ooperative or Hamsters.js) to simplify the communication with Web Workers and achieve the same goal.

The visual representation of a Reo circuit helps to understand the control-flow of a JavaScript application more easy. Especially compared to a JavaScript application using a complex orchestration of callback-based APIs. However, the JavaScript world is
moving more and more to promise-based APIs solving most of the problems caused by callbacks. Still, Reo has some advantages: Reo adds some nice properties to support discussing/reasoning about the control-flow of our JavaScript application. Also, Reo helps developers to think about reuse (of protocols) upfront.

Reo has the potential to introduce a library of reusable (language independent) Reo protocols (on the web). Imagine that we (or someone from the community) has already designed a Reo circuit that fits our need. It does not matter this protocol was used in another target language (e.g. Java or C) before. We can easily use this Reo circuit, link it to our own processes and compile it into JavaScript. This concept will perfectly fit into the open source community embraced by a lot of JavaScript developers.

We will conclude our evaluation with possible the biggest advantage of using DSLs like Reo: We can benefit from updates more easily. When the Reo-to-JavaScript compiler comes with a new version providing new features (e.g. improved performance or a runtime security fix) it is only a minor change to implement this update into our applications. We only need to recompile the protocol instead of adapting all our application manually.

### 10.3 Related work

This section describes some related work in the area of improving concurrent and asynchronous JavaScript programming.

<table>
<thead>
<tr>
<th>Program comprehension</th>
<th>Reuse of synchronization logic</th>
<th>Simplify utilizing Web Workers</th>
<th>Testability of synchronization logic</th>
<th>Inversion of control issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reo</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Promises</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Async/await keywords</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript superset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control flow libraries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web Worker libraries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novel JavaScript engines (features)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program comprehension tools (like Clematis and Theseus)</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Comparison of related work

Table 10.1 compares the related work described in this section to Reo based on the issues described in Section 4.1. With Reo we were able to address all those issues. However, by combining the alternatives discussed in this section we can address most issues as well. Where Reo can really make a difference is in the area of reusing synchronization logic and
the potential for improving the testability of synchronization logic (by applying formal verification and model checking).

**Evolving (supersets of) JavaScript** As described in Section 6.3, JavaScript itself is evolving so we can write asynchronous programs that look more like synchronous programs [15]. Promises (see Section 2.2.1) and generators (see Section 2.4.3) have arrived with ECMAScript 6, and async functions (see Section 6.3.2) are coming in a future ECMAScript version. Alternatively, some programming languages define a superset of JavaScript and have language features to ease asynchronous programming. TypeScript [39] is probably the most known of them and supports async functions already for some while.

**JavaScript libraries** There are a lot of (free) JavaScript libraries for managing asynchronous control flow (like async.js [49] and Q [50]) or simplify the communication with Web Workers (like Parallel.js [45], Operative [46] and Hamsters.js [47]). Those libraries are most often easier to apply than Reo, but do not encourage to separate computation code from programming synchronization/communication code.

**Transformation tools** As described in Section 6.3.4, there exist a number of transformation tools that enable code written in a synchronous style to be automatically transformed into an asynchronous style (like Continuation.js [51] and Streamline.js [41]). These tools require special marks that have to be manually inserted in the code for marking the beginning and end of the continuation. Philips, Laure, et al. [11] introduces a tool that does not require explicit annotations in the code to distinguish asynchronous function calls from synchronous function calls. This can be a source of inspiration when we want to implement our Reo transpiler for synchronous I/O operations mentioned in Section 6.3.5.

**Novel JavaScript engine (features)** As described in Section 2.3, JavaScript lacks support for shared memory parallelism. TigerQuoll [12] presents a novel parallel JavaScript engine allowing applications to exploit a mutable shared memory space. It would be very interesting for Reo, when programming models like this are adopted by popular JavaScript engines (for example to implement blocking I/O operations).

**Tools to enhance program comprehension** Clematis [13] and Theseus [14] are tools for helping developers understand complex asynchronous control flow in JavaScript code by capturing low-level interactions and visualizing those as higher-level behavioral models. Clematis and Theseus are reverse engineering tools with the main purpose to enhance program comprehension of an existing JavaScript application, while Reo is used to specify the control flow of a new JavaScript application.

**10.4 Future work**

This section contains proposals for future work.

**Simplify linking of JavaScript processes** There are many more improvements we could implement in the Reo designer to make JavaScript support more easy. We can implement support for importing processes by drag/dropping JavaScript files on the canvas. This features is already available for Java (as described in Section 3.2.2), but we did not implement this for JavaScript. Also, linking asynchronous APIs to a Reo circuit is a bit cumbersome, because you need to specify the meta-data of the component in the correct format. A more user friendly approach is to show a wizard or a list of well known (built-in browser) APIs.
**Run protocol in Web Worker**  The JavaScript runtime for Reo currently runs the generated protocol in the main (UI) thread. A possible optimization is to run the protocol also in a background thread (in a Web Worker). This would minimize the impact of Reo on the main (UI) thread. The processes would run in a sub worker in that case. We decided not to implement this feature for this thesis, because nested workers are not supported by all JavaScript engines.

**Error handling**  Another improvement for the JavaScript runtime for Reo is error handling. Currently we do not handle errors from asynchronous APIs connected to a Reo circuit. When such an API throws an error, the Reo application will just crash. We should define a generic way to handle those errors.

**Dynamic number of workers**  Currently it is not possible to instantiate a Reo circuit with a variable number of workers. This limited us to implement all features of the first reference application as described in Section 9.1.7. Implementing this feature would be a very nice topic of future work.

**Bundle/transpile generated code**  As described in Section 8.4, the Reo-to-JavaScript compiler currently generates multiple JavaScript files. One for each protocol, one for each component within the protocol and a main file to connect them all. To improve browser loading times it might be better to bundle all those files into one single file. This would make it also easier to embed a Reo circuit into another program. To improve browser support we can also transpile our JavaScript code to ECMAScript 5 as described in Section 5.7.

**Study use-cases**  We need to study more use-cases for Reo in actual (real world) applications to answer the question how useful Reo actually is in the development of real world JavaScript programs. Implementing more use-cases will answer the question if Reo makes it easier to understand the control flow of concurrent JavaScript applications more easy. Although we provided some preliminary insights on what the performance consequences of Reo are in JavaScript programs, we need to execute more (precise) performance tests/measurements.

**Formal verification and model checking**  Reo has a formal foundation, which enables formal analysis (like model checking [10]). This should make statically verifying a given protocol relatively easy. A very nice topic of future work is to find out if Reo can improve the testability of interaction- and synchronization logic of JavaScript applications by model checking.

**Support Java Futures in Reo runtime for Java**  Java has Futures, which are close to Promises in JavaScript, and allow similar patterns of code. Despite the fact that Futures are not woven into Java as Promises are into JavaScript, it is interesting to study if it is possible to link Java Futures directly to a Reo protocol using the same approach we can link asynchronous JavaScript APIs.
Bibliography

Scientific references


Books


Online references


[34] Babel - The compiler for writing next generation JavaScript. URL: https://babeljs.io/ (visited on 01/02/2017).

[35] browserify. Browserify lets you require('modules') in the browser by bundling up all of your dependencies. URL: http://browserify.org/ (visited on 01/02/2017).


[38] umdjs/umd: UMD (Universal Module Definition) patterns for JavaScript modules that work everywhere. URL: https://github.com/umdjs/umd (visited on 01/02/2017).
TypeScript - JavaScript that scales. URL: https://www.typescriptlang.org/ (visited on 01/02/2017).

Oni Labs: StratifiedJS. URL: http://onilabs.com/stratifiedjs (visited on 01/02/2017).


jQuery. URL: https://jquery.com/ (visited on 01/02/2017).

StringTemplate. URL: http://www.stringtemplate.org/ (visited on 01/02/2017).

Pedro Verruma. Javascript Web Workers Test v1.4.0. URL: http://pmav.eu/stuff/javascript-webworkers/ (visited on 02/01/2016).

Parallel.js: Parallel computing with Javascript. URL: https://parallel.js.org/ (visited on 01/02/2017).

padolsey/operative: Seamlessly create Web Workers. URL: https://github.com/padolsey/operative (visited on 01/02/2017).

Hamsters.js — Javascript Multithreading. URL: http://www.hamsters.io/ (visited on 01/02/2017).

OpenWeatherMap. Weather API. URL: http://openweathermap.org/api (visited on 02/01/2016).

colan/async: Async utilities for node and the browser. URL: https://github.com/colan/async (visited on 01/02/2017).

kriskowal/q: A promise library for JavaScript. URL: https://github.com/kriskowal/q (visited on 01/02/2017).

BYVoid/continuation: A compiler for JavaScript asynchronous Continuation-Passing Style transformation. URL: https://github.com/BYVoid/continuation (visited on 01/02/2017).