Towards Tierless Web Development without Tierless Languages

Laure Philips* Coen De Roover‡* Tom Van Cutsem* Wolfgang De Meuter*

* Software Languages Lab, Vrije Universiteit Brussel, Belgium

‡ Software Engineering Laboratory, Osaka University, Japan

lphilips, cderoove, tvcutsem, wdmeuter @vub.ac.be

Abstract

Tierless programming languages enable developing the typical server, client and database tiers of a web application as a single mono-linguistic program. This development style is in stark contrast to the current practice which requires combining multiple technologies and programming languages. A myriad of tierless programming languages has already been proposed, often featuring a JavaScript-like syntax. Instead of introducing yet another, we advocate that it should be possible to develop tierless web applications in existing general-purpose languages. This not only reduces the complexity that developers are exposed to, but also precludes the need for new development tools. We concretize this novel approach to tierless programming by discussing requirements on its future instantiations. We explore the design space of the program analysis for determining and the program transformation for realizing the tier split respectively. The former corresponds to new adaptations of an old familiar, program slicing, for tier splitting. The latter includes several strategies for handling cross-tier function calls and data accesses. Using a prototype instantiation for JavaScript, we demonstrate the feasibility of our approach on an example web application. We conclude with a discussion of open questions and challenges for future research.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Concurrent, distributed, and parallel languages; D.2.11 [*Software Architectures*]: Patterns (client/server)

General Terms Languages, Design

Keywords Tier splitting, Program slicing, Tierless Programming, JavaScript

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

Contemporary web development has become complex. There is an increasing demand for interactive features, collaboration between clients, support for offline functionality, etc. Realizing such advanced features in a traditional three-tier architecture requires developers to select and master a myriad of technologies. Each tier comes with its own technology stack. Examples include a query language for the database tier, PHP or Java for the server tier, and a combination of JavaScript, HTML and CSS for the client tier —often augmented with cross-tier technology for asynchronous communication such as Ajax and jQuery. It is up to the programmer to combine and align the different technology stacks. This might not only require a lot, but also rather complex glue code for contemporary web applications. For instance, to ensure the different data models of each tier are kept in sync.

Tierless programming languages aim to reduce this complexity. They enable developing a web application as a single mono-linguistic application, which renders its development akin to that of a desktop application. A preprocessor or the runtime of these languages realizes a split into a client, server and sometimes a database tier, where communication between the different tiers is handled transparently. The dynamically-typed functional language Hop [10] is an early example. It discerns code destined for the server and client tier based on developer-provided annotations at the level of individual expressions. The statically-typed functional languages Links [2] and Opa[1] require such annotations at the level of complete functions.

These approaches to tierless web development require an investment in novel and perhaps esoteric programming languages. More importantly, they require developers to annotate code meticulously with tier splitting information —a time-consuming and often error-prone process. This is particularly problematic given the general lack of tool support for these languages. Developers are on their own as far as understanding, testing, validating, debugging and refactoring tierless web applications is concerned. We therefore advocate to develop tierless web applications in a general-

[1] `http://www.opalang.org`

purpose language instead, such that its existing tool support can be leveraged. JavaScript, for instance, is already a common compilation target for the client and even some server tiers of the aforementioned tierless languages. This is because of its ubiquitous browser support and its successful entry on the server-side since the introduction of NODE.JS. We propose to take this trend one step further and develop tierless web applications in JavaScript itself.

The approach we envision requires a means to split an "ordinary" JavaScript program into client and server-side code. We identify an old familiar, program slicing [14], as the enabling technology for our vision. Developers are only required to adorn a minimum of code with `@client` and `@server` annotations. Program slicing will uncover the implicit dependencies between annotated code, thus determining the border along which a tierless program can be split in a server and client tier. Inconsistently annotated code can be warned about along the way. For example, client-side annotations do not make sense on code that performs an operation on the server-side database. The same goes for server-side annotations on code that accesses the browser's DOM. Conversely, existing development tools for the general-purpose language are oblivious to these annotations. They will therefore continue to treat the tierless web application as any single-tiered one. As such, our approach alleviates the need for additional tool support for understanding, testing, validating, debugging and refactoring web applications.

The concept of tier splitting is not novel. VOLTA [5], J-ORCHESTRA [11] and GWT [3], for instance, are capable of splitting tierless Java and C# programs into a *distributed application*. Our contribution over these "LAMP era"[2] approaches is the identification of open problems and possible solutions that are inherent to the "JavaScript era" with highly interactive, contemporary *web applications*.

2. Motivating Example

We motivate our approach using a web-based chat application. We describe the functional requirements of the chat application before scrutinizing a tierless implementation in a prototypical tierless language. A prototype instantiation of our approach will enable demonstrating an equivalent JavaScript implementation in Section 3.1.

2.1 A Web-based Chat Application

The client side of the web application is to let users join a chat room under a particular user name. Initially, every user has joined a `default` chat room under the name `guest`. The web browser should display a separate HTML paragraph for each message that has been shared with the users in a chat room. A "Send" button and input field will let users share such messages themselves. Finally, a "Name" button and input field will allow users to change their name in the current chat room. Callback functions associated with these buttons can perform the required communication between each client and the server.

The server-side of the web application is to maintain a mapping from users to chat rooms. To this end, it should listen for events originating from the callback functions in its clients. Chat messages can be broadcast to the users in a chat room through events as well. Modern web applications and frameworks (see e.g., Meteor in Section 7) typically achieve this through asynchronous AJAX-style calls or push-communication over web sockets (where the server pushes data without this being solicited by the client).

2.2 Tierless Implementation in a Prototypical Tierless Language

Listing 1 depicts an implementation of the chat application in Hop [10]. Hop is a dynamically-typed tierless programming language derived from Scheme. As such, its syntax is based on s-expressions. Line 14, for instance, invokes a function `<HTML>` on the result of an invocation of a `<BODY>` function. These are built-in functions that generate the corresponding HTML tags. In this case, an HTML page will be generated whenever a client connects to the "hello" service defined through the `define-service` special form on line 10.

HOP web services are evaluated on the server tier. Developers can use annotations to change the tier an individual expression of the service will be evaluated on. Annotation `$` causes an expression to be evaluated on the server tier, while annotation `~` causes an expression to be evaluated on the client tier. Note that these annotations can be nested arbitrarily. The `onclick` handler for the "Name" button on line 36, for instance, is executed on the client tier (first `~`). However, line 12 defined the input field for this name on the server tier. A typical Hop server program builds the HTML tree and ships it to the client. Therefore, the input field is defined on the server side, inside a `let` statement, so that it can be referenced in the listener for the name button, but also to embed it in the HTML tree (line 33). To retrieve the current value in that text field, we must escape from the client to the server tier using the `$` annotation. Having changed the name through the `set!` expression, we use the `with-hop` construct to invoke the server-defined function `join-room` from the client tier. Note that we have to annotate the reference to this function properly again.

Line 4 defines a server-side web service that, when invoked, broadcasts events of type "chat" to every client. Clients subscribe to these events using the listener defined on line 19 (i.e., one listener per client). The body of this listener appends a new paragraph for the newly received message to the `<DIV>` of existing messages. Note again that there is a `$` in front of the reference to the corresponding server-side variable.

The nesting of annotations can be confusing at times. One needs to keep track of the tier each referenced variable and function is defined on. Errors such as violating the scoping

[2] http://gigaom.com/cloud/node-js-and-the-javascript-age

```
(define nr_clients 0)

(define-service (chat msg)
  (hop-event-broadcast! "chat" msg))

(define-service (join-room name room)
   ;Store the update
  )

(define-service (hello)
  (let ((msg_in (<INPUT>))
        (name_in (<INPUT>))
        (msgs (<DIV>)))
  (<HTML>
    (<BODY>
     ~(define name "guest")
     ~(define chatroom "default")
     (set! nr_clients (+ 1 nr_clients))
     ~(add-event-listener!
            server
            "chat"
            (lambda (e)
               (dom-append-child! $msgs
                           (<P> (event-value e))))
            #t)
      msg_in
      (<BUTTON>
        :onclick
        ~(let ((msg $msg_in.value))
           (with-hop
             ($chat (string-append name  ": " msg))))
        "Send")
      name_in
      (<BUTTON>
        :onclick
        ~(let ((new_name $name_in.value))
           (set! name new_name)
           (with-hop
             ($join-room new_name chatroom)))
        "Change name!")
      msgs))))
```

Listing 1. Motivating Example in Hop

rules of a particular tier or using tier-specific primitives are easily made.

On the other hand, this example clearly illustrates how tierless programming reduces the complexity of developing web applications. Exposure to the underlying technology stack is reduced significantly. Developers can focus on the application logic rather than worry about non-negligible amounts of glue code for client-server communication, conversions from one data representation to the other, etc. . .

2.3 Towards a Tierless Implementation in a General-Purpose Language

We envision a different approach to tierless programming. Through program analysis and transformation, this approach should bring the benefits of the paradigm to existing general-purpose languages. We formulate the following requirements on instantiations of this approach:

R1. Support tierless programming in a general-purpose language Tierless programming currently requires investing in a tierless programming language and its accompanying tool support —at least to the extent such support exists. We believe that the state of the art in program analysis and transformation technology can enable tierless programming for existing general-purpose languages instead. Cross-tier, bi-directional communication should be supported in a seamless manner. For instance, by aligning it with the language's built-in procedural abstractions. Developers should be shielded from the accidental complexity that comes with the underlying technology stack (e.g., asynchronous AJAX-style calls and push-communication over web sockets).

R2. Compute a sound tier split from a minimal number of annotations Existing tierless programming languages rely on developers to demarcate the tier split using meticulously-specified annotations. This brings about the time-consuming and error-prone process of determining the server-side and client-side dependencies of each expression. The sometimes intricate nesting of these annotations (e.g., lines 36–39 in Listing 1) might prove difficult to maintain.

We therefore advocate computing the tier split through a program analysis instead. Although a small amount of annotations will always be required to initiate the analysis and to resolve uncertainties in its results, incorporating a state of the art analysis should minimize their incidence. Moreover, such an analysis will enable warning about incorrectly placed annotations. For instance, it could enable warning about server-side annotated functions that update a DOM element.

R3. Enable the use of existing development tools on tierless programs Existing general-purpose programming languages are supported by tools for understanding, testing, validating, debugging and refactoring programs. Our approach should enable using these tools on tierless programs developed in such a language. For instance, developers should be able to apply a rename refactoring consistently across all tiers. Provisions might have to be made for code that depends on libraries that are specific to a particular tier. For instance, validating or testing code that manipulates a DOM requires the actual DOM or a shadow.

R4. Support full-fledged JavaScript in the browser Contemporary web applications such as collaborative editors offer a high degree of interactivity. To the users, such applications appear as a "single page" to which elements are continuously added or removed from. These page changes are typically implemented as DOM manipulations through JavaScript libraries such as JQuery and Dojo. Any approach to tierless web development should therefore support the use of existing JavaScript libraries in a web application.

R5. Offline functionality and consistency strategies Being able to continue using a web application while disconnected is a feature that ranges from being useful (e.g. working on an airplane) to being critical (e.g. in disaster-relief scenarios where network connectivity is sparse). A common way of enabling offline functionality is to replicate (part of) the state that represents the application on the client. This way, the client has the required data locally until it reconnects. When changes can be made to this data, means to keep

this data consistent between the server and all (possibly disconnected) clients should be offered.

3. Overview of the Approach

We introduce an approach to tierless programming that fulfils the above requirements. To this end, it relies on program analysis and transformation for determining and realizing the tier split respectively. Figure 1 depicts an overview of the different phases in this tier splitting process. The process starts from a tierless program in a general-purpose language and results in tier-specific code (in the same language) ready to be deployed in a distributed setting. We now describe each phase and its input and output artefacts in an abstract manner. Section 4 will detail their instantiation in our prototype implementation.

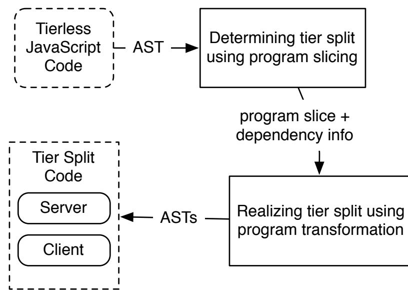

Figure 1. Process-centric overview of our approach.

```
1  s1   var nr_clients = 0;
2  Ds   /* @server */
3     {
4  s2    var db = [];
5  e1    var joinRoom = function (name,chatroom) {
6  s3         // add to mapping
7         }
8     }
9
10 Dc   /* @client */
11    {
12 s4   var name = 'guest';
13 s5   var chatroom = 'default';
14 s6   nr_clients += 1;
15 e2   var printMsg = function (msg) {
16 c1         print('msgs', read('msgs') + '\n' + msg);
17        }
18 c2   install('name_btn', 'click',
19 e3       function() {
20 s6           name = read('name_input');
21 c3           joinRoom(name, chatroom)
22       });
23 c4   install('send_btn', 'click',
24 e4      function () {
25 s7         var msg = read('msg');
26 c5         print('msg','');
27 c6         printMsg('me: ' + msg)
28 c7         broadcast(chatroom, name+': + 'msg);
29       });
30 c8   subscribe(chatroom,
31 e5       function (data) {
32 c9           printMsg(data)
33       })
34    }
```

Listing 2. Motivating Example in Tierless JavaScript

3.1 Tierless JavaScript Code

Our approach is applicable to any general-purpose language that supports distributed programming. It relies on annotations for developers to specify what tier a particular block of code belongs to. We will illustrate our approach using JavaScript. As JavaScript features no syntax for annotations, we require developers to nest tier annotations in the comments for a code block.[3]

Listing 2 depicts the tierless JavaScript code for the web-based chat application from Section 2.1. The code looks like regular JavaScript, except for the `@client` and `@server` annotations that reside within the comments. The web application can therefore be run, tested, debugged or refactored as any regular JavaScript program. Being tierless, however, its architecture corresponds roughly to the Hop implementation from Section 2.2. For instance, lines 19–22 install a callback handler for the "Name" button that enables users to join a chat room under a particular username. Pushing the button results in an invocation of the `joinRoom` function. Note that this function happens to reside in an `@server` block.

Line 19 installs this callback handler through a function `install(id, event, callback)`. We provide this function in a JavaScript library of *tierless primitives* that facilitates common operations in web applications such as manipulating the DOM and communicating between the users of the application. Other primitives used in our chat application are `print(id,text)` and `read(id)` for DOM manipulation, and `broadcast(type,message)` and `subscribe(type, callback)` for event-based communication. These tierless primitives abstract over the myriad of technologies that can be used to this end.[4]

In contrast to existing approaches, we only require developers to annotate a minimal seed for the tier split. The full split will be computed by analyzing the tierless program for cross-tier references and dependencies. Redundant annotations are allowed, but the consistency of all annotations will be verified.

3.2 Determining the Tier Split through Program Slicing

Using the developer-provided annotations, the second phase in our tier splitting process computes the border along which a tierless program can be split in a server and client tier. We discuss now how program slicing [14] can be adapted to this end. Traditional applications of program slicing are to be found within program comprehension and debugging, for instance to determine what parts of the program contributed to an erroneous value.

[3] A similar annotations-in-comments convention is followed by JsDoc and the Google Closure compiler, among others.

[4] However, future instantiations of our approach can always forego this library in favor of recognizing uses of the underlying technologies themselves.

3.2.1 Program Slicing

Informally, a backward program slice is a subset of an application that has a direct or indirect effect on the values computed at a given location. A program slice is therefore computed with respect to a slicing criterion, typically a line number and a set of variables. In general, slicing algorithms operate upon a program dependency graph: a directed graph of which the nodes correspond to statements and branch predicates, and of which edges correspond to data and control dependences. A statement is data dependent on another if values flow from the latter to the former. A statement is control dependent on a branch predicate if the outcome of the latter determines whether the former will be executed. For instance, there is a control dependency from every statement of a procedure to its entry point. Whole-program graphs include parameter binding edges from concrete arguments to formal parameters. Computing a slice then amounts to finding all nodes that can be reached backwards from the criterion. We can split a properly annotated tierless web application into a server and a client tier through slicing. The client and server slices are obtained using criteria that consist of all statements within a `@client` and a `@server` block respectively. Program slicing requires a small initial seed for the partitioning, and enables arbitrarily complex nestings of client and server expressions. It guarantees that all dependencies are included in the resulting slice, but also has other advantages such as the ability to filter out duplicated code but int he meantime preserves the original program's behavior.

3.2.2 Distributed Dependency Graph

The distributed nature of web applications requires some extensions to the traditional notion of a dependency graph. We extend it with so-called *distributed component* nodes that encapsulate a code snippet that is to be executed by a particular tier. Each statement within such a component has a control dependency on the component's entry point. Cross-tier data and control dependencies become *remote dependency* edges. Our extensions are akin to the C-nodes proposed by Mohapatra et al. [7] to represent the sending and receiving of messages between processes.

Figure 2 depicts a distributed program dependency graph (DPDG) for the code in Listing 2. For illustration purposes, we limit the graph to lines 1 to 22 from the code example. The mapping from line numbers to node numbers can also be seen in the listing. The distributed program dependency graph contains all the nodes and edges of a traditional program dependency graph. For instance, there is a data dependency from the variable declaration `s5` to its use as the second argument `a6` for the call at `c3`. In addition, the graph has distributed component nodes `Dc` and `Ds` for the client and server tiers respectively. A remote edge goes from the client-side call `c3` to the entry point `e1` of the server-side function that is called.

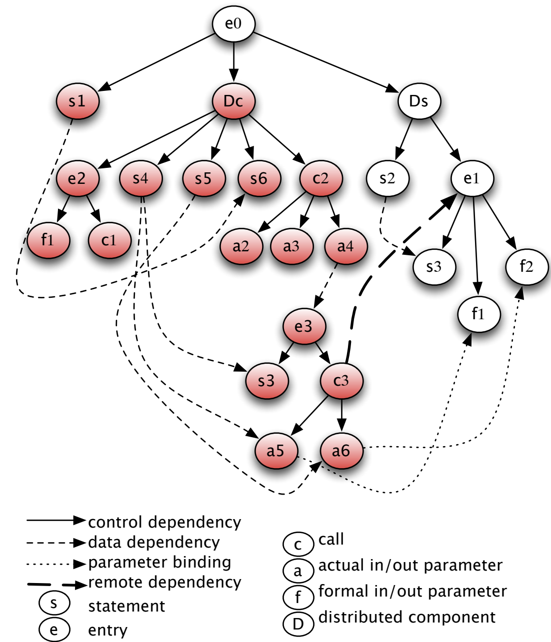

Figure 2. Distributed dependency graph for lines 1-22 of the tierless code in Listing 2. Highlighted nodes correspond to the slice computed along the tier split.

3.2.3 Slicing along the Tier Split

To separate the client tier from the server tier, we slice the program along a criterion that consists of all statements within their respective distributed component. For our motivating example, we slice on statements `s4`, `s5` and `s6,` call node `c2` (together with its arguments `a2`, `a3`, `a4`), and entry nodes `e2` and `e3`. This way we cover all statements in the distributed component and the slicing algorithm ensures us that all dependencies are included in the resulting tier split. Figure 2 highlights the slice obtained in this manner for the client tier.

Incorporating program slicing in our approach enables developers to sketch the tier split using coarse-grained annotations at the level of code blocks. We deem this less tedious and error-prone than meticulously specifying the split using fine-grained annotations at the level of individual expressions. Note, however, that the final tier split is more sophisticated than the initial developer-provided sketch. A function may even end up with a different body on the client tier and on the server tier.

3.3 Realizing the Tier Split through Program Transformations

Slicing a tierless program into tier-specific subsets does not yet produce deployable code. Actually realizing the tier split requires injecting distributed programming technology for the communication across tiers and between tier peers. The

remote edges in the program's distributed dependency graph can be consulted to this end.

However, the transformations required for cross-tier calls and references are both diverse and complex. What transformation to apply does not only depend on the chosen technology stack, but also on the tier a function (or data definition) is destined for and the tiers from which it is called (or referenced). We present the results of a case analysis below.

In the interest of generality, our case analysis does not yet commit to any particular distributed programming technology. Section 4 details the actual transformations incorporated in our prototype, including the intricacies brought forth by technology for asynchronous procedure calls and consistency management of replicated data.

3.3.1 Case Analysis for Cross-Tier Calls

In tierless programming, cross-tier communication is seamless through regular function calls. Table 1 summarizes when such a call should be transformed to a remote procedure call (or an equivalent service invocation). We distinguish the cases where the function is either defined on the server, defined on the client or defined in neither tier (i.e., resides outside of an annotated code block). Conversely, a function can be called only from the server tier, only from the client tier, from both tiers, or by neither tier.

Function	**CALL**			
	server-only	client-only	from both tiers	
DEF			server-side	client-side
server tier	same	RPC	same	RPC
client tier	RPC	same	RPC	same
shared	same	same	same	same

Table 1. Case analysis for transforming cross-tier calls.

When no specific transformation is required for the corresponding function call it is indicated in the table with "same". This is the case for functions that are only called from the tier they are defined on (i.e., all calls to the function are intra-tier). Functions that are shared by both tiers can be duplicated in the tiers where they are used (together with all their dependencies). The following special cases can be identified in the table:

1. Function defined on the server tier, called by the client tier (and possibly the server tier): the function call on the client should be replaced by a client-initiated remote procedure call. Depending on the chosen distributed communication technology, server-side calls to the same function might require a local copy of the original definition.
2. Function defined on the client tier, called by the server tier (and possibly the client tier): the function call on the server should be replaced by a server-initiated remote procedure call. This requires the distributed communication technology to support bi-directional communication (e.g., push-based communication over web sockets in our context). Depending on the specifics of this technology, client-side calls to the same function might require a local copy of the original definition.

Note that, in all of the above, "duplicating" a function definition might require a sophisticated transformation for applications that rely on the identity of its returned values.

3.3.2 Case Analysis for Cross-Tier References

Several strategies can be used when dealing with remote data in distributed programming. We could, for instance, transform cross-tier data manipulations to remote mutator and accessor calls. Alternatively, we could also replicate the data and keep the replicas consistent across the tiers and tier peers. In our setting, replication comes with the benefit of faster read and writes. It also does not preclude clients from using the web application offline. However, a consistency mechanism should be applied to guarantee an (eventually) consistent state of all tiers and tier peers.

Variable	**REFERENCE**			
	server-only	client-only	from both tiers	
DEF			server-side	client-side
server tier	same	R+C	R+C	
client tier	/	same	/	same
shared	same	R+C	R+C	

Table 2. Case analysis for transforming cross-tier references.

Table 2 summarizes our case analysis. A slash indicates that the corresponding combination is not allowed scoping-wise. "Same" indicates that no transformation is required. This is the case when data is defined on a certain tier and is used by the same tier. In these circumstances, the data can simply be defined and used in the scope of that tier. Entries of the form R+C indicate that data should be replicated with provisions for its consistency:

1. Data defined on the server tier, referenced by the client tier: a transformation should be applied that results in the data being replicated to the clients. All replicas should be kept consistent.
2. Data defined outside of a tier-specific code block (i.e., shared), referenced by the client or server tier (and possibly both): a transformation should be applied such that the involved tiers operate upon replicas of the shared data. If only the server tier references the shared data, we can simply move the definition to the server scope. No transformation of the references is required. However, if only the client tier references the shared data, we still need to transform the references to ensure each client operates upon a replica. The same transformation is required when the shared data is referenced from both tiers.

Note that in tierless programming, references from the server tier to a specific client cannot be expressed through scoping

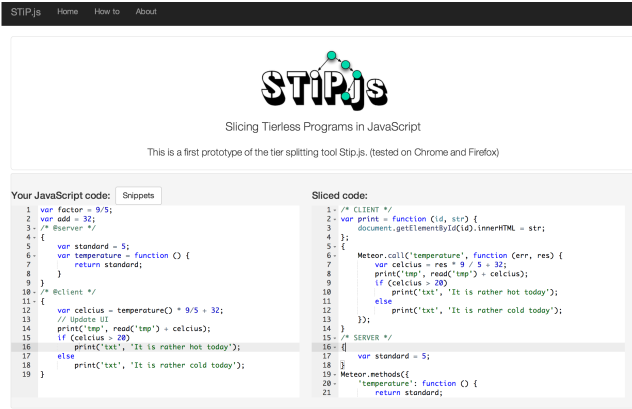

Figure 3. STiP.js in action: editor on tierless and final tier-split code.

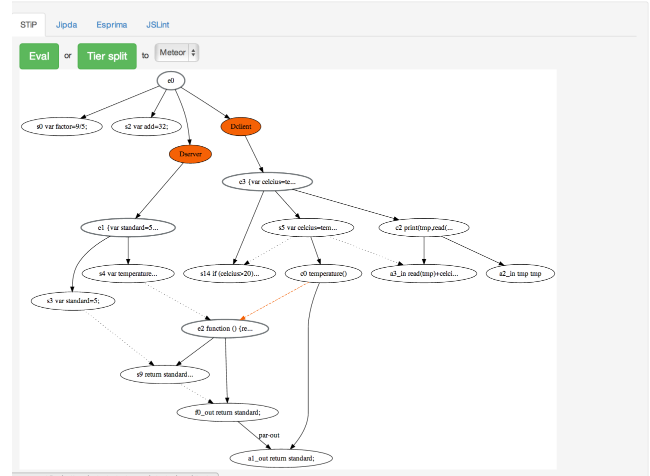

Figure 4. STiP.js in action: dependency graph used to determine the tier split for the code in Figure 3.

alone. Such cases require dedicated tierless primitives similar to those introduced in Section 3.1.

4. Implementation

To evaluate the feasibility of our approach, we implemented a prototype capable of splitting a tierless JavaScript program into a client and server tier. Our prototype implementation, called STiP.JS (Slicing TIerless JavaScript Programs), is freely available at `http://soft.vub.ac.be/~lphilips/stip`. This website provides example programs of which the distributed dependency graph (see Section 3.2.2) can be visualized and sliced interactively. Figure 4 depicts such a visualization for the example program depicted in Figure 3. The remainder of this section discusses the highlights of this implementation.

4.1 Abstract Interpretation of JavaScript to Uncover Control and Data Dependencies

Our tool constructs its distributed dependency graph from the inter-procedural control and data dependencies that exist between the expressions in the tierless JavaScript program. We compute these dependencies from a compile-time description of the possible states the program can transition to. This description is provided by the JIPDA[5] abstract interpreter framework for JavaScript [8]. We therefore owe our support for a fairly representative subset of JavaScript to JIPDA —including several features that are difficult to analyze statically such as higher-order functions with side-effects and prototype chains.

4.2 Distributed Programming in JavaScript

Once a program slice has been obtained along the tier split, the next step is to replace cross-tier calls by an appropriate remote procedure call. To this end, our prototype generates code for the METEOR[6] web application framework.

```
1  if (Meteor.isClient) {
2    Meteor.call('temperature', function(err, res) {
3      // Check first if error
4      var celcius = res * 9/5 + 32;
5      // Update UI
6      if (celcius > 20)
7        // Display 'Summer!'
8      else
9        // Display 'Winter!'
10   })
11 }
12 if (Meteor.isServer) {
13   Meteor.methods({
14     'temperature' : function() {
15       // Retrieve current temperature
16       return temp;
17     }
18   })
19 }
```

Listing 3. Cross-tier calls as explicit remote procedure calls in the METEOR framework.

Listing 3 illustrates how this framework supports asynchronous remote procedure calls. Line 14 registers an anonymous function as a remote procedure under the name `temperature`. This enables other framework users to call the function as depicted on line 2. The `Meteor.call` construct takes as its first argument the remote procedure's name, followed by the arguments for the call, and a two-argument callback function. The callback function will be invoked by the framework with the return value of the anonymous function and an error if it occurred. The depicted callback function updates the application's user interface upon receiving a temperature reading from the remote procedure.

Note that the METEOR framework already supports tierless programming in a limited sense. Indeed, the same code is executed on the `Meteor.isServer` and `Meteor.isClient` users of the framework. However, the run-time checks for these flags indicate that the framework does not perform any kind of tier splitting. As a result, pos-

[5] `https://github.com/jensnicolay/jipda`

[6] `http://www.meteor.com`

sibly sensitive server-specific code ends up with clients as well. Moreover, developers cannot use the language's built-in procedural abstractions for cross-tier calls. Listing 4 depicts a truly tierless version of the same code, as supported by our approach. Section 7 revisits METEOR from the perspective of related work.

4.3 Transforming Synchronous Cross-Tier Calls into Asynchronous Remote Calls

While the case analysis of Table 1 summarizes which cross-tier function calls require a transformation, it does not define whether a remote call should proceed in a synchronous or asynchronous manner. Our prototype implementation prefers asynchronous communication whenever possible. This way, the client tier does not have to halt its execution while waiting for the server to communicate an answer.

However, asynchronous communication brings about the problem of ensuring that the result of a call has been received at the moment it is required. For instance, when the result has to be used as an argument to another call. One way to achieve this is to transform the call asynchronously, but to include all of its dependent expressions in the body of the callback function. When the result of the call becomes available, the callback function is invoked with this result as input upon which the computation continues.

Listing 4 depicts the tierless JavaScript variant of the METEOR program in Listing 3. Note that the `temperature` function is now defined as a regular JavaScript function. The client block makes a local function call to this function and proceeds as if the result was returned immediately.

Our implementation transforms this tierless JavaScript to the METEOR code given in listing 3. To decide which expressions should be moved into the body of the callback function, STIP.JS uses the dependency graph. The program dependency graph for the tierless code reveals which expressions depend on the result of the function call. Because the variable declaration on line 10 of listing 4 depends on the return value of the function call, the declaration should be moved to the callback handler. All other statements that are control or data dependent on that statement, are also moved to the continuation. For every statement that is transferred to the continuation, all dependent statements should be transferred as well, and so on. Other statements that have no dependency on the result of the function call can be executed independently.

5. Evaluation

In this section we evaluate qualitatively whether our approach and its prototype instantiation fulfill the requirements from Section 2.3. To this end, we apply STIP.JS on the tierless JavaScript for the motivating example.

```
1  /* @server */
2  {
3    var temperature = function () {
4      // Retrieve current temperature
5      return temp;
6    }
7  }
8  /* @client */
9  {
10   var celcius = temperature() * 9/5 + 32;
11   // Update UI
12   if (celcius > 20)
13     // Display 'Summer!'
14   else
15     // Display 'Winter!'
16 }
```

Listing 4. Seamless cross-tier calls in tierless JavaScript.

5.1 Evaluation on Requirement 1

R1: Support tierless programming in a general-purpose language.

Our tierless implementation of the web-based chat application in Listing 2 is valid JavaScript. We do employ some tierless primitives for manipulating the DOM and cross-tier communication. These tierless primitives abstract over the myriad of technologies that can be used to this end. However, an implementation of these primitives is provided as a regular JavaScript library.

The tier annotations `@server` and `@client` reside within the comments for code blocks. This way, other development tools that are oblivious of these annotations can still be used. In contrast to existing approaches, we only require developers to annotate a minimal seed for the tier split. The full split is computed by analyzing the tierless program for cross-tier references and dependencies. Redundant annotations are allowed, but the consistency of all annotations will be verified.

5.2 Evaluation on Requirement 2

R2: Compute a sound tier split from a minimal number of annotations.

Listing 5 depicts the client and server tiers computed by our prototype for the motivating example. Note that the original call to the server-side function `joinRoom` has been replaced by the METEOR construct `Meteor.call` on the client-side (line 13). The server-side now defines this function as a METEOR method (line 33), rendering it available for clients to call. This is consistent with our case analysis in Section 3.3.

Variables `name` and `chatroom` are defined on the client tier and so no transformation took place. Data replication and synchronization, for the shared variable `nr_clients` from the motivating example, is currently not implemented, but we give a strategy in section 5.

The tierless primitives we provide also appear in the split code and their implementation should thus be included in the project as a library. While the primitives used for DOM

```
/* Client code
*  Requires: primitives, Meteor Streams package
*/
var chatroom = 'default';
var name = 'guest';

var printMsg = function(msg) {
  print('msgs', read('msgs' + '\n' + msg)
};

install('name_btn', 'click', function () {
  name = read('name_input');
  Meteor.call('joinRoom',name,chatroom);
});

install('send_btn', 'click', function() {
  var msg = read('msg');
  print('msg', ' ');
  printMsg('me: ' + msg);
  broadcast(defaultstream, name+ ': ' + msg);
});

subscribe(defaultstream, function(data) {
  printMsg(data);
});


/* Server code
*  Requires: Meteor Streams package
*/
var db = [];

Meteor.methods({
  joinRoom : function(name,chatroom) {
    // add to mapping
  }
});
```

Listing 5. The sliced programs in METEOR

```
var defaultstream = new Meteor.Stream('default');
var subscribe = function (stream, topic, fn) {
      stream.on(topic, fn);
};
var broadcast = function (stream, topic, data) {
      stream.emit(topic, data);
};
```

Listing 6. The communication primitives for Meteor

manipulation (`read`, `print` and `install`) are rather elementary, we discuss how the primitives for communication (`broadcast` and `subscribe`) are transformed to METEOR code.

To broadcast something to all clients, we utilize the Streams package of METEOR. It is basically a topic-based publish/subscribe protocol where clients subscribe to a certain topic using `stream.on(topic, callback)`. Publishing a message can be achieved by using `stream.emit(topic, data)`. This way of communication can be used to address only a subset of all clients or even one in particular. The code generation phase for METEOR thus also includes the code depicted in listing 6. Line 1 creates a stream for the topics used in broadcast and subscribe. The primitives then simply use the publish subscribe system from METEOR Streams.

5.3 Evaluation on Requirement 3

R3: Enable the use of existing development tools on tierless programs.

Because we support tierless programming in a general-purpose language, existing tools can be applied to the tierless JavaScript code. To demonstrate, we take two common development tools: a refactoring and a unit testing tool.

Refactoring Using the Eclipse IDE for JavaScript (version 4.2.2) (provided by the Web Tools Platform[7], version 1.3.0) we refactored the code from listing 2. First, we renamed the shared variable `nr_clients` to `nr_of_clients`. The rename refactoring replaces every reference to that variable with the new identifier. Because the tool is unaware of the tiers (this information is inside comments), the refactoring was completed successfully for each tier. Secondly, we refactored the name of the function `joinRoom` to `addToRoom`. The Eclipse refactoring was able to detect that there is a call to this function in the client-block and therefore correctly adjusted that call accordingly. Would we have tried to use a refactoring tool on a typical tiered application developed in different languages and technologies, refactorings on one tier would have completed successfully, but cross-tier refactorings would likely not be supported.

Unit testing The Mocha test framework [8] allows unit testing of JavaScript code. We implemented the small tierless JavaScript program depicted in Appendix A, Listing 8. Its server tier defines two functions to test whether a given number is a prime, and list all prime numbers up until a certain number. The client tier has an input field and a button. When clicked, the client asks the server if the input number is prime or not and displays the result.

In order to validate the program, we implemented unit tests for the server functions. These are depicted in Appendix A, Listing 9. As can be seen from the test code, no additional steps need to be taken to extract these functions from the server tier. The tests use the functions just as they were defined in a normal non-distributed JavaScript program.

This shows that the chosen approach, where tier-specific information is given inside comments, enables the code to be used by tools for that general-purpose language.

5.4 Evaluation on Requirement 4

R4: Support full-fledged JavaScript in the browser.

Our approach supports the use of client-side and server-side libraries in a web application. Its inter-procedural client and server slices are computed for the whole program, including all of its JavaScript libraries.

In practice, however, the STIP.JS prototype is limited by the precision of the abstract interpreter upon which it relies to compute the program's dependency graph. Its handling of

[7] `http://www.eclipse.org/webtools/`

[8] `http://visionmedia.github.io/mocha/`

```
/* Client code
*  Requires: CloudTypes library
*/
var nr_clients = state.get('nr_clients');
nr_clients.add(1);
...

/* Server code
*  Requires: CloudTypes library, */
CloudTypes.declare('nr_clients', CInt);
...
```

Listing 7. Cloud Types applied in tier split code

event handlers, in particular, is imprecise. Such imprecisions might lead to spurious branches in the derived dependency graph. Should these preclude computing a program slice unambiguously, we require additional annotations from the developer.

5.5 Evaluation on Requirement 5

R5: Offline functionality and consistency strategies. Implementing offline functionality requires the consistent application of strategies for replicating data and safeguarding the consistency of the replicas. We advocate the use of annotations as a means to express such strategies in a declarative manner, the implementation of which can then be taken care of by the runtime. This way, our approach facilitates implementing offline functionality in web applications. In fact, other cross-cutting concerns such as failure handling, security and persistence can be supported similarly.

Although the runtime of our prototype does not yet support these annotations, we can illustrate that it should be straightforward to add. Cloud Types [1] is a model for eventual consistency, developed in the TouchDevelop[12] language. Eventual consistency is consensus between the availability of the data and strong consistency of the replicas. Based on revision diagrams, the server keeps track of the main revision of the data, while the replicated revisions synchronize periodically with the main revision. Each of the Cloud Types has a predefined set of operations, next to the `set` operation. For cloud integers for example, the add operation is commutative and different adds can thus be executed incrementally on the revisions.

An open implementation of the model for JavaScript is available[9] and can therefore be incorporated in the tier split code. Listing 7 illustrates how this JavaScript library can be put to use to fulfill the offline functionality requirement. The original code used the variable `nr_clients`, that was declared on the shared level of the tierless code and used by the client annotated code. The variable is declared as a `CInt` (Cloud Integer) on line 10 and has the default value of 0. On the client side, the cloud type can be received using the `state.get` construct, providing the name of the cloud type (line 4). Incrementing the value is translated to the add operation on line 5. Please note that the setup code required by the Cloud Types library is omitted. This is boilerplate code and can easily be generated.

[9] `https://github.com/ticup/CloudTypes-paper`

6. Discussion

Our approach enables a tierless development style for complex web applications in a general-purpose programming language. This relieves developers from having to align different client-side and server-side technologies, but without requiring an investment in an esoteric tierless language that might be short-lived. More importantly, existing software engineering tools for the general-purpose language can be leveraged to develop, test, validate, debug and refactor the web application as a traditional tierless application. A minimal amount of annotations conveys the information necessary to perform the tier split automatically —of which the existing tools are oblivious.

Although we demonstrated through a prototype implementation that our approach is feasible for the motivating example, complete qualitative and quantitative validations are needed and will be subject of future work. This will require resolving several open questions and research challenges.

Leveraging tools in the presence of tier-specific code

Section 5 illustrates how a development and validation tools can be used on a tierless web application. However, further research is needed to see whether existing software engineering tools can cope with the tierless version of a web application that contains tier-specific code. For instance, code destined for the client-side tier might perform manipulations of the DOM maintained by a browser. Emulating the browser through libraries such as JSDOM [10] or ZOMBIE.JS [11] might still enable testing the tierless application as a whole on a NODE.JS server.

As a first step, we provide primitives for widely used tasks in web applications: DOM manipulation, communication between clients, etc. The actual implementation of these primitives differs for each output target. For instance, communication between clients is implemented differently for the METEOR framework than for a NODE.JS application. To be able to run the tierless program, a default implementation of these primitives should be provided as a library.

Failure Handling

Many things can go wrong in a distributed setting. Different applications may have to react differently to network-related errors. The code in listing 4 and its split version in listing 3 illustrate this already. The callback function in the split version takes an extra argument for a potential error, while there is no sign of error handling in the tierless version. A potential solution could be to enable developers to introduce custom error handlers through

[10] `https://github.com/tmpvar/jsdom`

[11] `http://zombie.labnotes.org/`

annotations. These annotations could reside at the tier-level indicating that all errors on this tier should be handled this way, or on the level of individual statements.

Tool support for annotations

Inconsistencies can arise when developers annotate tierless code. For instance, when a server-side function updates the DOM. Our distributed program dependency graph enables detecting and reporting such inconsistencies to the developer. Other combinations of annotation can prove to be ambiguous. For instance, when a function is passed aopes an argument to another that applies the given function on a different tier. Future tools, could ask additional annotations from the programmer to decide where the function should be executed. On the other hand, expressions that perform DOM manipulation or tier-specific code could moreover be recognized by the analysis and automatically classified as client-side expressions. Should this be inconsistent with the annotations provided by the programmer, future tools could ask to resolve the conflict.

In addition, these future tools could make the programmer conscious of cross-tier transformations; e.g. local function calls that become remote calls. Programmers can inspect and analyze how these transformations influence the resulting behavior (e.g. performance) of the application.

7. Related Work

Transformation-based approaches VOLTA [5], J-ORCHESTRA [11] and GWT [3] all advocate developing web applications in a single general-purpose language. Table 3 evaluates these approaches on the requirements from Section 2.3.

The discontinued VOLTA [5] tool is most closely related to our prototype. It enables developing distributed applications in a tierless manner. VOLTA automatically adds distributed constructs as required by developer-provided annotations. All languages that compile to the Common Intermediate Language of the .NET platform are supported, of which the client tier is compiled to JavaScript. As a consequence, existing tools can be reused. In contrast to our approach, VOLTA supports *very coarse-grained annotations only* (i.e., at the class-level), such that complex behavior such as a method that behaves differently on the client or server tier is harder to implement. Support for the *client tier is limited* in VOLTA, because its compiler cannot translate every class or function call to JavaScript code. This issue is solved by generating procedure calls from client to server to execute that particular piece of code on the server. This of course generates more network traffic. Also, bidirectional communication is not supported; *only the client can poll the server*. To the best of our knowledge, *data cannot be shared* between the tiers in VOLTA.

J-ORCHESTRA [11] is an automatic partitioning system for Java bytecode. Given tierless Java code and a distribution plan, it rewrites the code to a distributed application, using Java RMI to enable cross-tier communication. Through an XML configuration file the user must define sites (client and server) and classify each class of the system under these sites. Its mobile classes and synchronization mechanisms allow data to be shared between tiers. J-ORCHESTRA's partitioning thus *operates at a class-level granularity, almost entirely pre-determined* through a GUI or XML file. This is due to its reliance on a type-based analysis to detect references between classes in different partitions, in contrast to our use of transitive control and data dependencies between expressions. Program slicing requires a much smaller initial seed for the partitioning, and enables arbitrarily complex nestings of client and server expressions. Furthermore, our annotation-based handling of tier demarcation also extends to concerns such as consistency, ownership, and robustness. In J-ORCHESTRA, it is for example necessary to add failure handling manually in the generated bytecode. Because J-ORCHESTRA does not focus on web applications but on distributed applications, it has *no support for JavaScript whatsoever*.

Google's GWT [3] enables developing an entire web application in Java. To this end, it provides a Java implementation of a suite of reusable GUI components. GWT offers a development mode and a production mode. In the development mode, the application is executed from Java bytecode. In the production mode, all client-side code is compiled to JavaScript and HTML. In contrast to VOLTA, J-ORCHESTRA and our approach, *cross-tier calls require an explicit remote procedure calling construct* that is rather involved. Because of this, applications developed with GWT are not fully tierless. Consequently, not every kind of tool for Java can be reused. Therefore, GWT has its own unit testing, called GWT JUNIT, which can handle these explicit remote procedure calls and other framework specific constructs. GWT fully embraces JavaScript and defines how every data type can be converted between Java and JavaScript, thus enabling the use of external JavaScript libraries. In addition, classes can be shared between the client and server tiers. However, they get duplicated and are altered separately without synchronization.

Our contribution over these transformation-based approaches is the identification of open problems and possible solutions that are inherent to modern web applications. While some of the discussed approaches allow data replication to achieve a higher availability of the data, almost none of them consider conflict detection and resolution on the replicas. As illustrated in section 5, our approach can easily be extended in such a way that consistency strategies can be enabled automatically (as discussed in 3.3.2) or explicitly by means of annotations. We also strive to a way of programming where the programmer can use annotations to which other tools are ignorant and which capture all aspects of web development: e.g. failure handling should be handled

in the tierless code and not in the tier split code. This is not always the case for the other approaches discussed here, as a programmer is required to add specific failure handling in the generated code. STIP.JS also enables developers to implement a rich client, that can make use of all JavaScript libraries that are out in the wild.

	R1	R2	R3	R4	R5
Volta	✓	✓	✓	±	×
J-Orchestra	✓	✓	✓	×	✓
GWT	±	✓	±	✓	±
Koka	×	✓	×	×	×
Meteor	±	✓	±	✓	±
Stip.js	✓	✓	✓	✓	✓

Table 3. Comparison of related work based on requirements (defined in section 2.3)

Analysis-backed tierless programming languages KOKA [4] is a functional programming language in which the effects of a function are automatically inferred and checked. A `client` and `server` effect can be used for functions that are only to be called from the client-side and server-side respectively. The `pure` effect is used for functions that can be called by both. This enables the effect system to detect undesired cross-tier calls, giving the developer certain guarantees about how the application will be split. This guarantee comes at the cost of having to invest in a new language and its tool support. A commonality with our approach is KOKA's use of program analysis. The one in KOKA takes the form of a type and effect system rather than a control and data flow analysis. KOKA does not yet support shared state between the client and server tier. As can be seem from table 3, KOKA does not allow existing tools to be reused (R3), neither does it incorporate support for JavaScript libraries (R4) or offline functionality (R5), but it has its own exception system embedded in the language.

Frameworks for tierless programming METEOR is a tierless framework that allows programmers to write a web application in JavaScript. While a web application is written in one single language, the programmer is still required to explicitly use the framework's RPC constructs, etc. to add distributed behavior (requirement 1). It offers a state-of-the-art data replication mechanism, but does not detect inconsistencies nor solves them, hence requirement five is only fulfilled partially. METEOR decides at runtime which code should be executed by looking at the current executing environment. This requires the program to be transmitted in its entirety to the client, including all of its server-side code. Our prototype implementation produces sliced code that can be run by the framework, where server-side code will not end up at the client side. We only rely on the framework's RPC constructs to realize the required bi-directional communication between clients and the server. In the future, we will investigate alternative JavaScript libraries or frameworks to this end.

Program slicing for web applications While the notion of slicing a web application is not new, its use as the enabling technology for tierless programming is. More traditional applications of program slicing are to be found within program comprehension and debugging. The REWEB program slicer [9, 13], for instance, incorporates dependencies between PHP scripts and the HTML code they generate in the program dependency graph of a web application. It can be used to create an executable slice of the complete web application with respect to a slicing criterion related to a debugging task. The FIRECROW [6] tool supports slicing the client-side of a web application only. To this end, it extracts dependency information related to the DOM from recorded execution traces.

8. Conclusion

We have identified program slicing as the technology with the potential of enabling tierless programming in general-purpose programming languages. This relieves developers from having to align different client-side and server-side technologies, but without requiring an investment in a new tierless language. In our approach to tierless programming, web applications are developed as ordinary single-tiered programs using the existing tools for a general-purpose language. Once tested and validated, the single-tiered program is automatically split into server and client tiers. This requires a minimal amount of annotations from developers. Program slicing technology uncovers the implicit dependencies between annotated code, thus determining the border along which the single-tiered program can be split. To realize the tier split, shared state and function calls are replaced by the appropriate distributed programming constructs. Using a freely available prototype implementation for JavaScript, we have demonstrated the feasibility of our approach on a small, but representative tierless web application. Future work includes further qualitative and quantitative evaluation of the approach on other tierless programs.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. Laure Philips is supported by a doctoral scholarship granted by the Agency for Innovation by Science and Technology in Flanders, Belgium (IWT). This work has been supported, in part, by the Japan Society for the Promotion of Science, Kakenhi Kiban (S), No.25220003, and by the Osaka University Program for Promoting International Joint Research.

References

[1] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud Types for Eventual Consistency. In *ECOOP'12*, pages

283–307, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31056-0.

[2] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming Without Tiers. In *FMCO*. Springer-Verlag, 2006.

[3] F. Kereki. *Essential GWT: Building for the Web with Google Web Toolkit 2*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321705149, 9780321705143.

[4] D. Leijen. Koka: Programming with Row-Polymorphic Effect Types. Technical report, Microsoft Research, 2013.

[5] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing Distributed Applications by Recompiling. *IEEE Softw.*, 25 (5):53–59, Sept. 2008. ISSN 0740-7459.

[6] J. Maras, J. Carlson, and I. Crnkovic. Client-side web application slicing. In *ASE*, pages 504–507, 2011.

[7] D. P. Mohapatra, R. Mall, and R. Kumar. A Novel Approach for Dynamic Slicing of Distributed Object-oriented Programs. In *ICDCIT'04*, pages 304–309, Berlin, Heidelberg, 2004. Springer-Verlag. ISBN 3-540-24075-6, 978-3-540-24075-4.

[8] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter. Determining Coupling In JavaScript Using Object Type Inference. In *SCAM13*, 2013.

[9] F. Ricca and P. Tonella. Web application slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 148–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1189-9. .

[10] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a Language for Programming the Web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.

[11] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 178–204, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43759-2. URL `http://dl.acm.org/citation.cfm?id=646159.680022`.

[12] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 49–60, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. . URL `http://doi.acm.org/10.1145/2048237.2048245`.

[13] P. Tonella and F. Ricca. Web application slicing in presence of dynamic code generation. *Automated Software Engg.*, 12 (2):259–288, Apr. 2005. ISSN 0928-8910. .

[14] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10 (4):352–357, 1984.

A. Unit Tests for the Evaluation

```
1  /* @server */
2  {
3    var getPrimes = function (max) {
4      var sieve = [], i, j, primes = [];
5      for (i = 2; i <= max; ++i) {
6        if (!sieve[i]) {
7              primes.push(i);
8              for (j = i << 1; j <= max; j += i)
9                sieve[j] = true;
10         }
11       }
12   return primes;
13   }
14
15   var isPrime = function (n) {
16     if (isNaN(n) || !isFinite(n) || n%1 || n<2)
17       return false;
18     if (n%2==0)
19       return (n==2);
20     var m=Math.sqrt(n);
21     for (var i=3;i<=m;i+=2) {
22       if (n%i==0)
23         return false;
24     }
25     return true;
26   }
27
28 }
29
30 /* @client */
31 {
32   install('prime_btn', 'click', function () {
33       var res = isPrime( read('number') );
34       print('isprime', res);
35   });
36 }
37
38 module.exports.isPrime = isPrime;
39 module.exports.getPrimes = getPrimes;
```

Listing 8. Tierless prime web application

```
1  var assert = require("assert"),
2        isPrime = require('./primes').isPrime,
3        getPrimes = require('./primes').getPrimes;
4
5  suite('isPrime', function () {
6    test('isPrime returns if a number is prime or not',
          function () {
7      assert.equal(true, isPrime(37));
8      assert.equal(false, isPrime(42));
9    });
10
11   test('zero and one are not prime numbers', function
          () {
12     assert.equal(false, isPrime(0));
13     assert.equal(false, isPrime(1));
14   });
15
16 });
17 suite('getPrimes', function () {
18   test('getPrimes returns all primes until given
         number', function () {
19     assert.equal(0, getPrimes(1).length);
20     assert.equal(1, getPrimes(2).length);
21     assert.equal(25, getPrimes(100).length);
22
23   });
24 })
```

Listing 9. Mocha unit test for listing 8