Developing JavaScript Applications in Eiffel

Master Thesis

By: Alexandru Ioan Dima
Supervised by: Prof. Dr. Bertrand Meyer
Dr. Martin Nordio
Christian Estler

Student Number: 09-934-928

Abstract

For numerous reasons, ranging from economical concerns such as distribution costs to security concerns (latest bug fixes), software projects are nowadays moving from native applications to web applications. Rich client applications are enabled by JavaScript, the programming language of the browser. However, JavaScript makes it easy to introduce errors because it is dynamically and weakly typed, because it is not compiled and because different browsers implement the specifications differently.

This thesis presents an automatic translator that transforms Eiffel programs to JavaScript programs. The translator supports the most important Eiffel features, including agents, contracts, exception handling, multiple inheritance, and once routines, bringing the engineering power of object-oriented programming practices and tools, together with contracts and void-safety to web applications. The applicability of the translator has been demonstrated with several case studies, including one where a full-fledged web-based source code editor was implemented.

Acknowledgements

I would like to thank my supervisors Dr. Martin Nordio and Christian Estler for their continuous support, trust and valuable feedback. My deepest gratitude goes to Prof. Dr. Bertrand Meyer for being my mentor, for his inspiring lectures and for giving me the opportunity to write the master thesis at the Chair of Software Engineering.

I consider myself fortunate and privileged to have Dr. Erich Gamma's support, helping me find the topic for this thesis and shaping my work through the valuable discussions we have had. Special thanks are due to Michael Schneider, Johannes Rieken and Dr. Dirk Baeumer for their kind support, precious advice and great feedback.

Especially, I thank Dr. Marius Minea for his great influence in my decision to pursue a Masters Degree at ETH. Last, but not least, I want to thank my family which supported me during my whole time at ETH.

Contents

Chapter 1

Introduction

1.1 Motivation

Today's software projects are increasingly moving from native applications to web applications. A web application is a program that is downloaded and executed by a web browser. Popular examples are email clients, map applications or social networks. They have several advantages such as no installation required (they run in the user's browser), always up to date (users always fetch the latest version of the application from the web server), centralised data and lower maintenance costs.

The most popular programming language of the browser is JavaScript[1]. Although it brings flexibility and power, programming with JavaScript introduces several problems. The language is dynamically and weakly typed and everything runs together in the same global scope, thus making it hard to define interfaces. Moreover, since JavaScript is not compiled and different browsers implement the specifications differently, it is easy for programmers to introduce errors.

Alternatives to JavaScript have been proposed. For example, browser plugins such as Adobe Flash [3] for ActionScript, Microsoft Silverlight [34] for .NET languages or Java Applets [4] for the Java language have been developed in order to enable programmers to execute code in the browser and to avoid writing JavaScript. Lack of adoption of these technologies [32, 33], performance or security issues [2, 11, 31] introduce risks in using these solutions. Even though some of these browser plugins provide APIs to a programmer which are not available from JavaScript, with the development and adoption of HTML5 [14], JavaScript gains even more ground.

[1] JavaScript is used throughout this thesis to refer to ECMAScript [1]

On the other side, object-oriented programming languages have demonstrated their power, flexibility and maintainability [20]. Developing programs in object-oriented languages such as C#, Eiffel, or Java is simpler.

The goal of this thesis is to design and implement an automatic translator that takes Eiffel programs [22] and produces JavaScript applications. This will enable developers to benefit from the advantages of web applications without giving up the engineering power of object-oriented programming practices and tools.

1.2 The CloudStudio Project

This thesis directly contributes to CloudStudio [28], a project which aims to develop a web-based development environment. The motivation of CloudStudio lies in the challenges introduced in distributed projects [21, 27, 30], where teams collaborate in a geographically distributed setting. Such distribution introduces new challenges [25, 30], for example how to design an API, how to write requirement documents or how to manage a project.

The CloudStudio project is a result of the experiences from a course on distributed software development, DOSE [26, 29], taught at ETH Zurich. During the course, software projects are implemented in a collaborative fashion by students from several universities in Asia, South America, and Europe. A lack of integrated tools that support distributed software engineering has been identified during the course.

The web-based Integrated Development Environment (IDE), developed in the CloudStudio project, addresses this void. Besides collaboration and project management tools, the IDE implements an awareness system that allows a developer to view the changes introduced by other team members in real time, thus helping developers to detect conflicts and problems earlier. This feature is unobtrusive since the user can select what changes from which developer to display. For example, developer A can choose whether to display or not the changes introduced by developer B. Thus, if the code introduced by B does not compile, A can choose to ignore B's changes, continuing his work.

1.3 Contributions

The main contribution of this thesis is an automatic translator that transforms Eiffel programs to JavaScript programs. The translator supports the most important Eiffel language features, including agents, contracts, excep-

tion handling, multiple inheritance, and once routines. Two case studies have been developed to demonstrate the applicability of the translator.

One of them is a web-based source code editor. The editor code is written in Eiffel and gets translated to JavaScript. Besides being able to run as a stand-alone tool, it has also been integrated to the CloudStudio IDE and serves now as the default source code editor.

This thesis is structured as follows: Chapter 2 presents the main challenges of translating Eiffel to JavaScript. Chapter 3 describes the implementation of the translator. Case studies are described in Chapter 4. Finally, related work and conclusions are presented in Chapter 5 and Chapter 6 respectively.

Chapter 2

From Eiffel to JavaScript

This chapter presents how Eiffel source code can be translated to JavaScript source code. It describes the challenges encountered when translating Eiffel concepts which do not have a JavaScript counter-part.

This chapter will also introduce a JavaScript object, `runtime`, that implements the most common operations with objects, handling class declarations, run-time type checks and some utility functions.

Furthermore, methodologies for using native browser objects from Eiffel source code and for translating source code which uses EiffelBase are presented.

2.1 Brief Overview of JavaScript

JavaScript is an object-oriented prototypal programming language [1]. Classes in JavaScript are not like in other object-oriented languages as C#, Eiffel or Java because there are no blueprints for instantiating objects. Instead, objects can be created using a literal notation or using other objects as their prototypes; **objects inherit from objects**.

2.1.1 Objects in JavaScript

Listing 2.1 shows how an object can be declared, created and used in JavaScript. In this example, objects are created using a *constructor* [1], a function called with the `new` keyword (line 16). The function `Student` is called with the passed parameters (`id=6,name="John"`) and a new object containing all the properties defined in `Student.prototype` is created and usable with the keyword `this`. Moreover, because the `Student` function contains no return statement, the object denoted by `this` is returned by default [1].

The example also shows how to define default values for attributes and how to add methods to a prototype. It is also important to mention that referring to the current object's properties is done exclusively with the **this** keyword[1].

Listing 2.1: Defining an object in JavaScript

```
1  // constructor for a Student object
2  function Student (id, name) {
3    if (id) { this.id = id; }
4    if (name) { this.name = name; }
5  }
6
7  // adding a default value for the attributes
8  Student.prototype.id = 0;
9  Student.prototype.name = "";
10
11 // adding a method
12 Student.prototype.learn = function () {
13   ...
14 };
15
16 var stud = new Student(6,"John");
17 console.info (stud instanceof Student);
18 console.info (stud instanceof Object);
```

The last lines show how the **instanceof** operator may be used to test objects against constructor functions. The output to the console is **true**, **true**. The latter shows that objects in JavaScript inherit by default from the `Object` type.

2.1.2 Inheritance in JavaScript

Listing 2.2 shows how inheritance can be achieved in JavaScript, by using other objects as prototypes. In order to express the fact that `PhDStudent` inherits from `Student` the instruction on line 6 changes `PhDStudent`'s prototype from the default (a new `Object`) and sets it to a new `Student`. The call also motivates the two **if** statements on lines 3 - 4 from Listing 2.1, the constructor is called without parameters.

[1] In other languages like C# or Java, qualifying current object's attributes or method calls with **this** is optional.

Listing 2.2: Inheritance in JavaScript

```
1  // constructor for a PhDStudent object
2  function PhDStudent (id, name) {
3    Student.call (this, id, name);
4  }
5
6  PhDStudent.prototype = new Student();
7
8  // adding a new method
9  PhDStudent.prototype.writeThesis = function() {
10   ...
11 }
12
13 var PhDstud = new PhDStudent(5,"Mary");
14 console.info (PhDstud instanceof PhDStudent);
15 console.info (PhDstud instanceof Student);
16 console.info (PhDstud instanceof Object);
```

The call to the constructor `Student` from line 3 is for initialization purposes and it is not compulsory in order for the inheritance to work, serving as an example to how static calls to a function are done in JavaScript. Fig. 2.1 shows the prototype chain for the created objects.

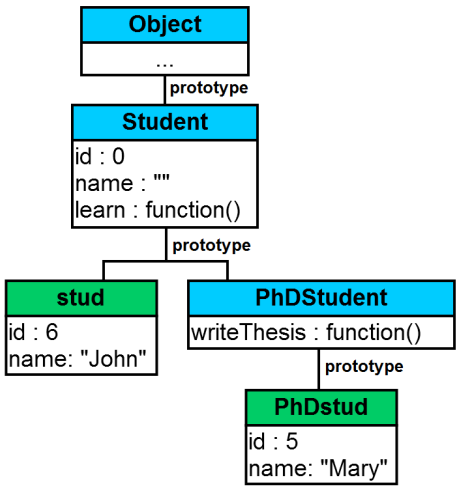

Fig. 2.1: Objects and their prototypes in JavaScript

The output to the console is `true`, `true`, `true`, thus showing how the `instanceof` operator travels up the prototype chain of an object until it finds the constructor function being tested for or until it finds the `Object` prototype.

2.1.3 Inheriting directly from other objects

This section describes how inheritance may be achieved in another fashion, which doesn't require *constructor* functions or explicit assignments of the prototype by the programmer. This method is used in Section 2.2.1 to emulate multiple inheritance in JavaScript.

Crockford [5] (see Listing 2.3) defines `Object.create`, which creates a new object with a specified prototype. An extended version of `Object.create` has been introduced in the 5th Edition of ECMAScript [1].

Listing 2.3: Crockford's `Object.create` [5]

```
1 if (typeof Object.create !== 'function') {
2     Object.create = function (o) {
3         function F() {}
4         F.prototype = o;
5         return new F();
6     };
7 }
```

The objects created in this manner can be augmented with additional properties and they can be used themselves as prototypes for other objects. However, since there are no explicitly declared *constructors*, `instanceof` can no longer be used.

Listing 2.4: Using literal objects and `Object.create`

```
1 var Student = {
2   id : 0,
3   name : "",
4   initStudent : function (id, name) {
5     this.id = id;
6     this.name = name;
7   },
8   learn : function () {
9     ...
10  }
11 };
12 var PhDStudent = Object.create(Student);
13 PhDStudent.writeThesis = function() {
14   ...
15 }
16 var stud = Object.create(Student);
17 stud.initStudent(6, "John");
18
19 var PhDstud = Object.create(PhDStudent);
20 PhDstud.initStudent(5, "Mary");
```

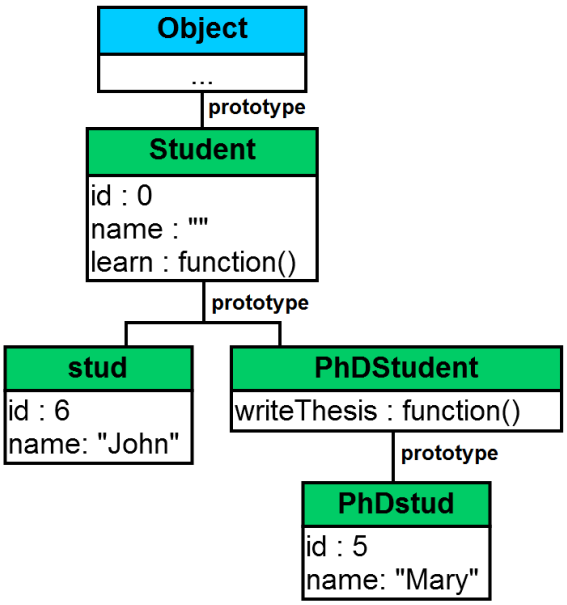

Fig. 2.2: Inheriting from other objects in JavaScript

Listing 2.4 shows how the `Object.create` mechanism can be used to achieve inheritance. In this example, the `Student` object is defined using JavaScript's object literal notation and it is the prototype for `PhDStudent` and `stud`. `PhDStudent` is augmented with a new method and it is the prototype for `PhDstud`. A convention is used to execute initialization (`initStudent`).

The only disadvantage to using this approach is that the `instanceof` operator can no longer be used, since there are no *constructors* to test against (i.e. `Student` and `PhDStudent` are objects and cannot be used in an `instanceof` check). Thus, object type checks must be implemented manually.

2.2 Translating Eiffel language features

Translating Eiffel source code is done automatically with a translator (see Chapter 3) that uses the Eiffel compiler [7] and can handle a large number of Eiffel features. Since both Eiffel and JavaScript are high-level programming languages, a large percentage of simple Eiffel instructions and expressions have immediate equivalents in JavaScript.

This section will focus only on the Eiffel concepts which are more complex or which need to be simulated to some extent in JavaScript: multiple inheritance, redefining and renaming inherited features, once routines, agents, rescue clauses and contracts.

2.2.1 Multiple inheritance

Since inheritance in JavaScript is designed to allow objects to inherit from only one other object, multiple inheritance can be achieved by creating an object which collects attributes and methods from multiple sources and by using it as a prototype when creating other objects.

Listing 2.5: Multiple inheritance in JavaScript

```
1  var Teacher = {
2    teach : function() { ... }
3  };
4  var Student = {
5    learn : function() { ... }
6  };
7  var PhDStudent = {
8    teach : Teacher.teach,
9    learn : Student.learn,
10   writeThesis : function() { ... }
11 };
12 var s = Object.create(PhDStudent);
```

Listing 2.5 shows how multiple inheritance can be emulated by creating a "stitched" object containing all the attributes and methods from the multiple objects being inherited, `PhDStudent` in this case, with methods pointing directly to the ones from its parents. Then, this object can be used as a prototype in order to create new objects (see line 12).

The `runtime` object

Since supporting Eiffel's multiple inheritance is a main goal, defining objects and then using them as prototypes via `Object.create` has been chosen. Creating a "stitched" object with methods and attributes from different sources must only be done once for each Eiffel class, and then it can be used as a prototype.

However, doing the "stitching" manually is time-consuming, error-prone and hard to maintain. Therefore, a JavaScript helper object has been introduced: the `runtime`. Its purpose is to hide the implementation details of this "stitching" and to provide a clean and easy to use interface for defining objects and achieving multiple inheritance, acting as a *facade* [12].

Declaring an object

Since we are translating Eiffel classes, the following method to declare a new class refers to the class concept as it is in Eiffel.

`runtime.declare : function(name, parents, decl)`

- `name` – String with the name of the class being declared.
- `parents` – An array containing the parents of the declared class. Each item in this list contains an object with the following fields:
  - `class_name` – String with the name of the parent class
  - optional `renaming` – An object with property-value pairs corresponding to initial - renamed feature's name pairs.
  - optional `redefining` – An array containing the feature names of features being redefined
- `decl` – A literal object containing the feature definitions of the declared class.

The `runtime.declare` method creates a new object containing all the features (methods and properties) from `decl` and then adds, one by one, all the features from each parent class, following the renaming and redefining rules. This object is then stored in an internal cache and will be used as a prototype when creating new instances of the class (via `Object.create`), or when other classes inherit from it.

The `runtime.declare` method returns a *constructor* function for the declared class (see Listing 2.6, line 5) so that the newly defined class can be instantiated with the `new` keyword. However, when called, the *constructor* instantiates a new object with `Object.create` and then returns the new object (Listing 2.6, line 10).

Listing 2.6: `runtime.declare` structure

```
1  runtime.declare = function (name, parents, decl) {
2    var class_prototype = decl;
3    // Inherit features from parents to class_prototype
4    ...
5    return function () {
6      var result = Object.create (class_prototype);
7      // Call the appropiate construction feature
8      ...
9      // Return the newly created object
10     return result;
11   };
12 };
```

Listing 2.7 shows how the `runtime` object can be used to achieve multiple inheritance. The code snippet shows how the "stitching" of methods and attributes from the parent classes is done automatically by the `runtime`.

Listing 2.7: Multiple inheritance with the `runtime` helper in JavaScript

```
1  var Teacher = runtime.declare("Teacher", [],{
2    teach : function() { ... }
3  });
4  var Student = runtime.declare("Student", [], {
5    learn : function() { ... }
6  });
7  var PhDStudent = runtime.declare("PhDStudent", [
8    {class_name:"Teacher"},
9    {class_name:"Student"}
10 ], {
11   writeThesis : function() { ... }
12 });
13 var s = new PhDStudent();
```

`Teacher` and `Student` do not inherit from other classes (as the empty arrays on lines 1 and 4 show), while `PhDStudent` inherits from both `Teacher` and `Student`. `Teacher`, `Student` and `PhDStudent` are references to *constructors* and that is why objects can be created with the `new` keyword (see line 13).

Eiffel types

An Eiffel type is represented in JavaScript by an object with the following properties:

- optional `attached` – `true` or `false`, depending on whether the type is declared attached or not
- `name` – the name of the class
- optional `generics` – an array containing the actual types for generic classes.

Here are a few examples of Eiffel types and the JavaScript objects representing them:

- `A` ⟹ `{name:"A"}`
- `attached A` ⟹ `{attached: true, name:"A"}`

- `A[B]` ⟹ `{name:"A", generics: [{name:"B"}]}`
- `A[attached B]` ⟹
  `{name:"A", generics:[{attached: true, name:"B"}]}`
- `A[B,C]` ⟹ `{name:"A", generics:[{name:"B"},{name:"C"}]}`

Constructing objects

When a class is declared using the `runtime`, a *constructor* is returned. The returned *constructor* may be used in a variety of ways:

- When no creation feature is called, the constructor is used without parameters:
  `create {A}` ⟹ `new A()`
- When a creation feature is called, its name is a parameter in the constructor call. The constructor automatically calls the creation feature:
  `create {A}.make` ⟹ `new A("make")`
- When a creation feature is called with parameters, they are added after the feature name in the constructor call:
  `create {A}.make(x,y)` ⟹ `new A("make", x, y)`
- When a local variable or attribute is created:
  `create a.make(x,y)` ⟹ `a = new A("make", x, y)`
- When a class with generics is instantiated, the first parameter in the constructor is an array containing objects describing the actual types for the generics. This list is stored in the `$generics` property of the newly created object:
  `create {A[B]}.make(x,y)` ⟹ `new A([{name:"B"}],"make",x,y)`

Object tests

As discussed in Section 2.1.3, the `instanceof` operator can't be used with objects created with the `runtime`. Therefore, the `runtime` object implements the helper method `runtime.inherits(obj, type)`.

The first parameter accepted by this method is a JavaScript object created with a constructor produced by the `runtime` and the second one is an object representing an Eiffel type.

The method uses information collected when the object's class was declared (from which parents the class inherits) and information collected when the object was created (if it has generics).

2.2.2 Redefining & Renaming

One of the interesting problems encountered while developing the translator and the object representation is the support for Eiffel's feature redefining and renaming. Listing 2.8 shows an example where class `B` inherits from `A`, redefining `A.g` and renaming `A.f` to `old_f`. Furthermore, `B` defines a new feature named `f` which coincidentally has the same signature as `A.f`. The signature of `B.f` and `A.f` need not be the same, as there is no relation between them. However, `B.g` needs to have the same signature as `A.g`, since the former is overriding the latter.

Listing 2.8: Redefining and renaming in Eiffel

```
1  class A
2  feature
3    f: STRING do Result := "A.f" end
4    g: STRING do Result := "A.g" end
5  end
6
7  class B
8    inherit
9      A
10       redefine g
11       rename f as old_f
12     end
13 feature
14   f: STRING do Result := "B.f" end
15   g: STRING do Result := "B.g" end
16
17   test
18     local
19       b: B; a: A;
20       r1, r2: STRING
21     do
22       create b;
23       a := b;
24       r1 := b.f + "," + a.f
25       r2 := b.g + "," + a.g
26     end
27 end
```

Running the above code and executing feature `test` will result in `r1` being `"B.f,A.f"` and `r2` being `"B.g,B.g"`. Since `b` is assigned to `a` on line 23, `a` and `b` hold the same object at run-time. However, `b.f` calls feature `f` from class `B` and `a.f` calls feature `f` from class `A`, as shown in `r1`. This is because feature calls in Eiffel depend also on the target's declared type, not only on the target's run-time type and on the called feature's name. However, calls in

JavaScript depend only on the run-time target type and the method name.

In order to solve this problem, Eiffel feature names are translated to fully qualified method names (i.e. unique in the system): each class in the system receives an unique identifier, which is appended to its feature names. Assuming A has id 1 and B has id 2, the translation of the previous example to JavaScript looks similar to Listing 2.9.

Listing 2.9: Redefining and renaming translated to JavaScript

```
1  var A = runtime.declare("A", [], {
2    f_1 : function () { return "A.f"; }
3    g_1 : function () { return "A.g"; }
4  });
5
6  var B = runtime.declare("B", [
7    {class_name:"A", renaming:{"f":"old_f"}, redefining:["g"]}
8  ], {
9    f_2 : function () { return "B.f"; }
10   g_2 : function () { return "B.g"; }
11   test_2 : function () {
12     var b, a, r1, r2;
13     b = new B(); a = b;
14     r1 = b.f_2() + "," + a.f_1();
15     r2 = b.g_2() + "," + a.g_1();
16   }
17 });
```

The call to `b.f` has been translated to `b.f_2` and the call to `a.f` has been translated to `a.f_1`, thus disambiguating the calls and resolving the problem. The downside to using unique method names is that native JavaScript overriding can no longer be used to implement feature redefining: the call to `b.g` is translated to `b.g_2` and the call to `a.g` to `a.g_1`. However, in both cases, the same feature must be called, feature `g` written in B.

This is solved by the `runtime`; when inheriting a feature which is redefined, its implementation is replaced with a proxy function which redirects the call to te redefining feature. This is shown in Fig. 2.3, where feature `B.g_1` has been defined as a proxy to `B.g_2`. This approach is modular and allows for multiple redefinitions of the same feature: should a third class inherit from B and wish to redefine `B.g`, it is only `B.g_2` which shall be overridden with a method that calls the new feature.

Fig. 2.3 also shows that there is no prototypal relationship between A and B.

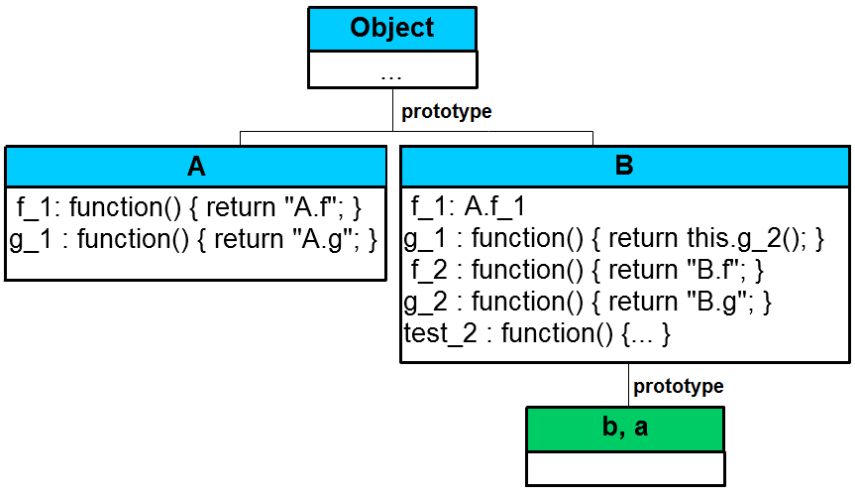

Fig. 2.3: Objects and their prototypes in JavaScript

2.2.3 Once routines

A once routine is a routine which is executed only the first time it is called. If the routine has a return value, subsequent calls return the result from the first execution. If it has no return value, subsequent calls have no effect [22].

Listing 2.10 shows an example of using once routines in Eiffel. Routine `b` of class `C` only executes the first time it is called, creating a new object of type `B` and assigning it to the result. After the first call, further calls return the same object.

Listing 2.10: Once routines in Eiffel

```
1 class
2   C
3 feature -- Access
4   b: B
5     once
6       --| Other code here
7       create Result
8     end
9 end
```

The same behaviour in JavaScript can be obtained with a self invoking function which returns a closure, like in Listing 2.11. The function declared on line 2 is self-invoking and will execute immediately, even before the `runtime.declare` call (since it is part of a parameter). Executing the outer function immediately means that the inner function, declared on line 4 *closes over* (i.e. captures) the `$cached` and `$executed` variables and is assigned to `b`.

Listing 2.11: Once routines in JavaScript

```
1 var C = runtime.declare("C", [], {
2   b : (function () {
3     var $cached = null, $executed = false;
4     return function () {
5       if (!$executed) {
6         $executed = true;
7         // Other code here
8         $cached = new B();
9       }
10      return $cached;
11    };
12  })() // self invoking because of ()
13 });
```

The `if` statement in the inner function (line 5) guards that the body of the translated routine only executes the first time it is called and then stores the `Result` in `$cached`. Subsequent calls do not execute the code inside the `if` statement and proceed directly to return the stored `Result`.

2.2.4 Agents

Agents [22] in Eiffel are first class citizens, they represent operations as run-time objects. Agents can be handled as any other object, they may be passed as parameters, they can be assigned to variables and so on. Agents may be inlined or created from an object's feature and they may contain open or closed arguments.

Listing 2.12 presents how agents may be used in Eiffel. The inline agents created on lines 2 - 5 are assigned to different positions in the list and on line 8, one is retrieved and called.

Listing 2.12: Inline agents in Eiffel

```
1 from i := 1 until i > 10 loop
2   x[i] := agent(a, b: INTEGER): INTEGER
3     do
4       Result := a + b
5     end (?, i)
6   i := i + 1
7 end
8 check x[2].item([1]) = 3 end
```

In the example, a list of agents is created, `x`; each `x[i]` contains a procedure with one open argument, `a`, and one closed argument, `b`, that has the

value `i`, for its entire lifetime. Therefore, when called with a parameter `a`, each `x[i]` returns `a + i`.

A direct equivalent to agents in JavaScript are functions. Listing 2.13 shows how agents are translated to functions. The outer function on line 2 is self-invoking; it executes for each iteration in the loop, each time with a different value for `b`. The inner function on line 3 is returned and it *closes* over `b` (i.e. it captures a different `b` each time).

Listing 2.13: Inline agents in JavaScript

```
1 for (i=1; i<=10; i++) {
2   x[i] = (function (b) {
3       return function (a) {
4         return a + b;
5       };
6   }(i)); // self invoking with i as a parameter
7 }
8 if(!x[2](1) === 3)) { throw "Assertion does not hold."; }
```

2.2.5 Rescue clauses

Exceptions in Eiffel represent contract violations and they may be triggered whenever a contract is broken, an assertion does not hold or an unexpected error is encountered [22]. Such exceptions may be handled in a special part of a procedure, namely a `rescue` block. If the code path in the `rescue` block contains a `retry` instruction, the routine is re-executed. If not, the routine *fails* and the exception is propagated to the routine caller.

Listing 2.14: Rescue example in Eiffel

```
1  rescue_example
2    local
3      retry_count: INTEGER
4    do
5      --| Risky code here
6    rescue
7      --| Exception handling code here
8      if retry_count < 10 then
9        retry_count := retry_count + 1
10       retry
11     end
12   end
```

Listing 2.14 shows a feature, `rescue_example`, which contains a `rescue` block. This particular example is written such that the body of the feature

may be executed a maximum of 10 times. The two comments, on line 5 and 7 represent placeholders for code.

Listing 2.15 shows the translation of feature `rescue_example` to JavaScript and reveals the intrinsic repetitive nature of a feature which has a `rescue` block. A local variable, `$retry` is automatically introduced by the translator, which is always set to `false` at the beginning of each iteration (see line 4) and which controls the repetition of the feature's body. Eiffel's `retry` instruction is translated to `$retry = true` (see line 11).

Listing 2.15: Rescue example in JavaScript

```
1  rescue_example : function() {
2    var $retry = true, retry_count = 0;
3    while ($retry) {
4      $retry = false;
5      try {
6        // Risky code here
7      } catch ($err) {
8        // Exception handling code here
9        if (retry_count < 10) {
10         retry_count = retry_count + 1;
11         $retry = true;
12       }
13       if (!$retry) { throw $err; }
14     }
15   }
16 }
```

A JavaScript `try - catch` construct is generated; the feature's body is wrapped around a `try` and the `rescue` body is inserted in the `catch`. Line 13 shows how the exception is re-thrown if the `retry` instruction is not called from the `rescue` block.

2.2.6 Contracts

Eiffel has native support for contracts [18, 19, 22] and they represent a key feature of the Eiffel language. Contracts are expressed as preconditions, post-conditions and class invariants. Since JavaScript does not support such constructs at language level, they are represented in JavaScript as manual assertions (`if` statements).

Listing 2.16 shows a class implementing a queue and one of its routines, `push`. The class in the example has an invariant, labeled `valid` which states that the number of elements is always smaller or equal to the queue's capacity. This invariant is evaluated before and after every call a client class makes to one of its features.

Listing 2.16: Using contracts in Eiffel

```
class
  QUEUE [G]

feature -- Access
  count, capacity: INTEGER

  push (element: G)
    require
      has_space: count < capacity
    do
      ...
    ensure
      count_increased: count = old(count) + 1
    end

invariant
  valid: count <= capacity
```

Feature `push` has a precondition, requiring that there is still space left to accommodate for the new element and a postcondition ensuring that the new element count is equal to the old count plus one. The specifications are not complete, but they serve as good examples to show how contracts may be used in Eiffel.

Listing 2.17: Translation of contracts to JavaScript

```
QUEUE = runtime.declare("QUEUE", [], {
  count : 0,
  capacity : 0,
  push : function (element) {
    if (!(this.count < this.capacity)) {
      throw "Precondition does not hold.";
    }
    var $old1 = this.count;
    ...
    if (!(this.count = $old1 + 1)) {
      throw "Postcondition does not hold.";
    }
  },
  $invariant : function() {
    if (!(this.count <= this.capacity)) {
      throw "Invariant valid does not hold.";
    }
  }
});
```

Listing 2.17 shows how contracts are translated to JavaScript. Like described in Section 2.2.5, when contracts are not respected, an exception is thrown. Preconditions, postconditions and invariants therefore throw exceptions when they do not hold.

Preconditions are checked at the beginning of methods, before any other logic has been executed (line 5). In order to deal with `old` expressions, which refer to the values of expressions *before* executing any instruction in the routine's body, the translator automatically introduces local variables which hold the value of the `old` expressions (line 8). These local variables are then substituted in the postcondition check (line 10), which is always located at the end of the method's body.

The invariant is translated to a method, `$invariant`, which doesn't have any arguments and is always called before and after any feature call. Listing 2.18 shows a client of the `QUEUE` class which calls routine `push`. Listing 2.19 shows how feature calls in JavaScript are always preceded and followed by calls to the object's invariant.

Listing 2.18: A queue client in Eiffel

```
1 local
2   q : QUEUE [INTEGER]
3 do
4   ...
5   q.push (2)
6   ...
7 end
```

Listing 2.19: A queue client in JavaScript

```
1 var q;
2 ...
3 q.$invariant();
4 q.push(2);
5 q.$invariant();
6 ...
```

2.3 Using native browser objects from Eiffel

When executed inside a web browser, global objects[2] are available to JavaScript programs. They enable the programs to interact with the user and the browser API. One of the most important global objects is the `window` object, through which JavaScript programs can access the current `document` and work with the nodes composing the Document Object Model (DOM) Tree or with the Cascading Style Sheets (CSS) rules of the web page. Commonly, these objects are used to implement a user interface in a web application.

One of the problems tackled by this thesis is exposing these native browser objects to Eiffel programmers, such that interaction with them can be achieved straight from Eiffel source code. To allow for this interaction, a stubbing methodology has been devised: **In order to use a native JavaScript object from Eiffel source code, a special Eiffel class, called a *stub*, must be created**.

This section shows how a stub can be defined, what rules to keep in mind when writing their external features and how special directives can be used to influence the translator's behavior for a certain class. Section 3.2 presents the Eiffel library which contains stubs for native browser objects.

2.3.1 Stubs

A stub usually describes a JavaScript object, therefore all its features are external and they represent a method or a property in the corresponding JavaScript object. The name of a stub's feature is decoupled from the name of the JavaScript property through the use of the **`alias`** part of the external. This decoupling allows to maintain the generally accepted coding style of Eiffel, where "_" is used in feature names to separate different words (as opposed to JavaScript, where camel case is the norm).

Section 2.3.2 discusses and shows how the **`alias`** part of externals may be used in various scenarios. Stubs are used by the translator only at compile time and all the calls to a stub's features are inlined and replaced with the **`alias`** part of the external feature.

Listing 2.20 shows the interface of one native object available when running JavaScript in a browser: the `History` object [14].

[2]The specification of JavaScript [1] is independent from the specification of the different objects available in a browser [14, 39, 40, 41, 42]

Listing 2.20: IDL Definition of the History object [14]

```
interface History {
  readonly attribute long length;
  void go(long delta);
  void back();
  void forward();
  void pushState(any data, DOMString title, DOMString url);
};
```

Listing 2.21 shows the Eiffel stub corresponding to the `History` object and it shows the relation between Eiffel features and the available JavaScript attributes and methods. The **`javascript`** entry in the note list at the beginning of the class represents a translator directive and offers a vital piece of information to the translator: the `JS_HISTORY` class is a stub for a native JavaScript object named `History`.

Listing 2.21: Eiffel stub for the History object

```
note
  javascript: "NativeStub:History"

class
  JS_HISTORY

feature -- Basic Operation

  length: INTEGER
    external "JS" alias "length" end

  go (a_delta: INTEGER)
    external "JS" alias "go($a_delta)" end

  back
    external "JS" alias "back()" end

  forward
    external "JS" alias "forward()" end

  push_state (a_data: ANY; a_title: STRING; a_url: STRING)
    external "JS" alias "pushState($a_data, $a_title, $a_url)"
        end
end
```

It is important to mention that the existence of the native browser objects has not been hard-coded in the translator, they have been described through the use of stubs, in a library (see Section 3.2), since the interfaces for native browser objects change as specifications evolve over time.

2.3.2 Translating externals

Externals, as seen in the previous section, are very useful because they empower Eiffel programmers to use objects and concepts from JavaScript and the browser. This section explains and discusses the rules for the translation of externals.

The translator uses the `alias` part of an external feature and then applies the following algorithm in order to produce a JavaScript call:

Listing 2.22: Algorithm for translating external calls

```
1  isStatic ←false
2  result ← alias
3  if result.startsWith("#") then
4      result.removeHead(1)
5      isStatic ← true
6  end if
7  if result.contains("$TARGET") then
8      result.replaceAll("$TARGET", callTarget)
9      isStatic ← true
10 end if
11 for all arg_i : arguments do
12     result.replaceAll("$arg_i", callParameter_i)
13 end for
14 if not isStatic then
15     result.prepend(".")
16     result.prepend(callTarget)
17 end if
```

The above algorithm shows how an external feature's arguments or the call target may be used inside externals. The following examples clarify how the algorithm functions:

a) **A simple example**. The call target is maintained by the translator and the feature invoked is substituted with the string found in the `alias` section of the external.

b) **Using arguments**. The feature's arguments can be used in the external by prepending a `$` to their name. All occurrences are substituted with the actual call's parameters.

c) **Using the call target**. Sometimes, the generated JavaScript will not have the form `target.method(..)`, and therefore `$TARGET` may be used inside the `alias` section.

d) It is possible to use `$TARGET` or a feature's arguments multiple times inside the **alias**.

e) Sometimes `$TARGET` isn't used, but a static call is still desired. This is achieved by beginning the **alias** with `#`.

	Eiffel Source		Generated JavaScript
a)	`as_lower: attached STRING` `  external "JS" alias "toLowerCase()" end`		
	`my_str.as_lower`	⟹	`my_str.toLowerCase()`
b)	`at alias "@" (i: INTEGER): CHARACTER` `  external "JS" alias "charAt($i-1)" end`		
	`my_str.at (some_index)`	⟹	`my_str.charAt(some_index-1)`
c)	`abs: INTEGER` `  external "JS" alias "Math.abs($TARGET)" end`		
	`my_int.abs`	⟹	`Math.abs(my_int)`
d)	`sign: INTEGER` `  external "JS" alias "($TARGET>0?1:($TARGET<0?-1:0))" end`		
	`my_real.sign`	⟹	`my_real>0?1:(my_real<0?-1:0)`
e)	`max_value: REAL` `  external "JS" alias "#Number.MAX_VALUE" end`		
	`my_real.max_value`	⟹	`Number.MAX_VALUE`

2.3.3 Special translator directives

A clear distinction has been made between the translator and the rules used at translation-time. To allow for better maintainability, as much information as possible has been defined outside of the translator. For example, the translator intrinsically *knows* how to translate an Eiffel string constant to JavaScript, but it has no hard-coded knowledge about the `STRING` class features.

Due to this fact, a methodology to feed external information into the translator has been devised. Classes are handled in different ways by the translator if the note list of the class contains an entry with the name **javascript**. Here are the four special types of classes recognized by the translator and how they influence the translation process:

Native Stubs

Native Stubs are stubs for objects available when running JavaScript in a browser, for example. The translator does not generate any code for classes which have the entry `javascript:"NativeStub"` in the note list. All features must be externals; object tests for such classes work correctly if the name of the native JavaScript object is present in the note value and the `instanceof` operator is used to translate them.

Simple Stubs

Sometimes, the programmer may want to write a class in JavaScript and then use it from Eiffel. A possible use case is when dealing with non-standard browser operations, such as copy-pasting, which are implemented differently in every major browser. The note entry `javascript:"Stub"` denotes an Eiffel class which describes the object written in JavaScript. All features must be externals. No code is generated for it and object tests are handled by `runtime.inherits`.

EiffelBase

The note entry `javascript:"EiffelBase"` marks a class which contains a JavaScript equivalent for one or more EiffelBase classes. Usually, features are not externals and code is generated for the class. Section 2.4 discusses and shows in detail how using EiffelBase is possible from source code that gets translated to JavaScript.

EiffelBase Native Stubs

The entry `javascript:"EiffelBaseNativeStub"` in the note list denotes a special kind of JavaScript equivalent for one or more EiffelBase classes, namely a native JavaScript object. This entry is used only for the EiffelBase classes presented in Section 2.4.3. All features must be externals; no code is generated for these classes.

2.4 “Translating” EiffelBase

To be adoptable by already existing projects with a large Eiffel code base it is important that existing source code can be translated to JavaScript without modification.

Therefore, it is critical to handle source code using classes from EiffelBase, “a library of fundamental structures and algorithms covering the basics of

computing" [10]. EiffelBase is written mostly in Eiffel with the exception of some special built-in features and some external C constructs which "fill gaps".

This section presents how the tool translates source code which uses EiffelBase classes and why I have chosen to redirect EiffelBase feature calls to JavaScript equivalents.

2.4.1 Discussion

Before describing the actual devised solution, since EiffelBase is mostly written in Eiffel, one may argue that a good solution would have been to translate the entire EiffelBase library to JavaScript and then replace the C constructs which filled in the gaps with equivalent JavaScript constructs which would have filled in the same gaps with the same logic. Although this solution might seem advantageous, in reality it presents a number of problems.

The biggest problem is that the implementation of some crucial EiffelBase classes are written with the target (C in this case) in mind. On one side, EiffelBase's `STRING` and `ARRAY` use class `SPECIAL` as their data model – a continuous memory zone –, while on the other side, JavaScript supports both strings and arrays natively, offering a wide range of methods and operations [1].

A difference in the level of abstraction offered by the two target languages (C and JavaScript) gives birth to another problem: direct memory access, pointer arithmetic, etc. are available in C, while JavaScript is at the same level of abstraction as Eiffel, offering a prototypal object model, garbage collection and so on. For example, since JavaScript supports random position insertion, deletion or retrieval for arrays, arrays are an equivalent for EiffelBase's `ARRAYED_LIST`. Another similar example is the case of EiffelBase's `HASH_TABLE` which has a direct equivalent in JavaScript, objects. Objects in JavaScript consist of pairs of attribute names and values or function names and functions.

Therefore, translating EiffelBase to JavaScript would add an unnecessary layer of abstraction and would not make use of JavaScript's language features. Moreover, doing such a translation would treat JavaScript more as a Virtual Machine (VM) for Eiffel than a language, impacting performance, since JavaScript's base data structures are optimized in JavaScript VMs [15, 38].

Furthermore, due to time constraints, I have aimed at a solution which allows an incremental definition of EiffelBase classes and features, as it is outside the scope of this thesis to support the entire EiffelBase library. My solution takes advantage of JavaScript's dynamic nature: when translating

Eiffel to JavaScript, **the translator requires that there are JavaScript equivalents only for the EiffelBase classes and features that the programmer actually uses**.

2.4.2 Redirecting EiffelBase calls

My solution consists of redirecting feature calls to EiffelBase classes to JavaScript equivalents at translation-time. Let `C.f` be a notation for the feature named `f` from class `C`.

We define μ(`C`,`f`), the redirection of feature `f` from class `C`:

$$\mu(\texttt{C}, \texttt{f}) = \begin{cases} \texttt{C.f} & \text{if C is not an EiffelBase class} \\ \texttt{B.f} & \text{if C is an EiffelBase class and B is a class that has an "EiffelBase" javascript note entry listing C} \\ \texttt{EIFFEL\_C.f} & \text{otherwise} \end{cases}$$

where `EIFFEL_C` is the class obtained by prepending `"EIFFEL_"` to `C`'s name. If `EIFFEL_C` is not found in the known universe, an error is generated.

The above function shows that feature redirection is performed only for EiffelBase classes and that the translator, by default, looks for EiffelBase equivalents by prepending `"EIFFEL_"` to their names (e.g. `EIFFEL_INTEGER` for `INTEGER`, `EIFFEL_LIST` for `LIST`, etc.). However, the translator can be directed to different equivalents through the use of the **`javascript`** entry in the note list, where EiffelBase classes can be listed (see Section 2.4.3 for an example).

An interesting discussion is what part of a feature call to use for this redirection. To familiarize the reader further with the issue, let's analyze a concrete example which uses EiffelBase. Listing 2.23 shows a code snippet from a class providing a prime number test, with the first 1000 prime numbers precomputed in `known_primes`.

Listing 2.23: Simple EiffelBase feature call example

```
1  class
2    PRIMES_PROVIDER
3  feature -- Basic Operation
4    is_prime (a_number: INTEGER): BOOLEAN
5        -- Is `a_number' a prime number?
6      do
7          -- Test to see if in precomputed set
8        if a_number < 1000 then
9          Result := known_primes.has (a_number)
10       else
```

```
11          ...
12        end
13      end
14 feature {NONE} -- Implementation
15   known_primes: SET[INTEGER]
16        -- A set of all prime numbers to 1000.
17 end
```

Ignoring the translation problems associated with booleans and integers, an interesting problem is how to redirect the call on line 9. Feature `has` called on line 9 is written in `CHAIN` and is inherited by `SET`. Even so, the call is not redirected to μ(`CHAIN`,`has`), but it is redirected to μ(`SET`,`has`), because **the tool redirects based on the call target's type**.

This allows to define EiffelBase class equivalents in JavaScript incrementally and independently of EiffelBase's class hierarchy. In this case the tool requires the equivalent for `SET` and not for both `SET` and `CHAIN`. Thus, the process of supporting EiffelBase classes and creating the JavaScript equivalents is significantly simplified.

Inheritance may still be used in the EiffelBase JavaScript equivalent classes. The redirection methodology only requires that a feature with the same name is present in the equivalent JavaScript class, that feature may very well be inherited from some other class.

2.4.3 EiffelBase and native JavaScript types

Some of EiffelBase's classes have obvious JavaScript native type equivalents. For performance reasons, instead of creating wrappers around native types and always boxing and unboxing, a selected number of classes from EiffelBase get translated straight to native JavaScript types. The native types available in JavaScript are: `Undefined`, `Null`, `Boolean`, `String`, `Number` and `Object` [1]. JavaScript's `Array` inherits from `Object`.

Listing 2.24: Features from the JavaScript equivalent for EiffelBase REAL

```
1 note
2   javascript: "EiffelBaseNativeStub: REAL, REAL_32, REAL_64"
3
4 class EIFFEL_REAL
5
6 inherit
7   ANY redefine out end
8
9 feature -- Basic Operation
10
11   divisible (other: INTEGER): BOOLEAN
```

```
12      external "JS" alias "($TARGET%%$other===0)" end
13
14    max (other: REAL): REAL
15      external "JS" alias "Math.max($TARGET,$other)" end
16
17    max_value: REAL
18      external "JS" alias "#Number.MAX_VALUE" end
19
20    out: attached STRING
21      external "JS" alias "($TARGET).toString()" end
22
23  end
```

The translator maps the following EiffelBase classes to native JavaScript types:

- `BOOLEAN` maps to JavaScript's `Boolean` type.
- `CHARACTER` maps to JavaScript's `String` object. At run-time a character and a one character string have the same representation, thus producing false-positive object tests.
- `NATURAL`, `INTEGER` and `REAL` map to JavaScript's `Number` type. Again, at run-time there is no way to differentiate between a floating point variable which holds an integer and an integer variable. Listing 2.24 shows a few of the equivalent features for EiffelBase `REAL` class. The note on line 2 instructs the translator to redirect all calls to features from classes `REAL`, `REAL_32` and `REAL_64` to features with the same name of `EIFFEL_REAL`.
- `STRING` maps to JavaScript's `String` type.
- `FUNCTION` and `PROCEDURE` map to JavaScript's `Function`.
- `ARRAY` maps to JavaScript's `Array` object. Unlike arrays in other languages[3], arrays in Eiffel allow programmers to have a variable lower index bound. Therefore, the JavaScript arrays representing an Eiffel array hold at index 0 the array's `lower` property.

Element	a_l	a_{l+1}	...	a_{u-1}	a_u
Eiffel index	l	$l+1$	...	$u-1$	u
JavaScript index	1	2	...	$u-l$	$u-l+1$

[3]Arrays in Java, C, C# are 0-based, i.e. 0 is the hard-coded lowest index bound.

This mapping may sometimes lead to interesting externals, as a code snippet from EIFFEL_ARRAY, presented in Listing 2.25, shows:

- `lower`: the value is stored at index 0 in the JavaScript array.
- `count`: the number of elements in the JavaScript array minus 1, representing the element at index 0 (`lower`).
- since `count = upper - lower + 1` $\Longrightarrow$ `upper = count + lower - 1`, which explains the external for `upper`.
- working with Eiffel indices always requires the translation by `lower -1` to the left, as can be seen in features `at` and `put`.

Listing 2.25: Features from the equivalent for EiffelBase ARRAY

```
1  note
2    javascript  : "EiffelBaseNativeStub: ARRAY"
3  class
4    EIFFEL_ARRAY [G]
5
6  feature -- Basic Operation
7
8    lower: INTEGER
9      external "C" alias "$TARGET[0]" end
10
11   count: INTEGER
12     external "C" alias "($TARGET.length-1)" end
13
14   upper: INTEGER
15     external "C" alias "($TARGET.length+$TARGET[0]-2)" end
16
17   at alias "@" (i: INTEGER): G
18     external "C" alias "$TARGET[$i-$TARGET[0]+1]" end
19
20   put (v: G; i: INTEGER)
21     external "C" alias "$TARGET[$i-$TARGET[0]+1] = $v" end
22
23 end
```

2.4.4 Special dispatched calls

The mapping of some EiffelBase classes to native JavaScript types and objects raises an interesting problem: **variables declared as ancestors of some EiffelBase classes may be both objects or primitive values at runtime**.

Listing 2.26 shows a simple implementation of the Bubble Sort Algorithm. The notable fact about this implementation is that the BUBBLE_SORT class is generic and its generic is constrained to be a descendant of EiffelBase's COMPARABLE.

Listing 2.26: A Bubble Sort Implementation in Eiffel

```
1  class
2    BUBBLE_SORT [G <- COMPARABLE]
3
4  feature -- Basic Operation
5    sort (lst: LIST[G])
6        -- Sort `lst' ascending
7      local
8        i: INTEGER
9        sorted: BOOLEAN
10       temp: G
11     do
12       from sorted := false until sorted loop
13         from sorted := true; i := 1 until i > lst.count loop
14           if lst[i-1].is_greater (lst[i]) then
15             sorted := false
16             temp := lst[i-1];
17             lst[i-1] := lst[i];
18             lst[i] := temp
19           end
20           i := i + 1
21         end
22       end
23     end
24 end
```

At run-time, G can be a user-written class inheriting from COMPARABLE. In this case, translating the call to `is_greater` on line 14 is no different than translating any other call to a feature. However, the problem appears if G is an EiffelBase class which gets translated to a native JavaScript type, such as STRING or INTEGER[4] because these native types do not have a method named `is_greater`.

Due to this fact, a helper method has been added to the `runtime` object to help dispatch calls to ancestors of EiffelBase classes which are translated to native JavaScript types. Therefore, the call on line 14 is translated to a call to `runtime.special_dispatch`, as in Listing 2.27.

[4]In EiffelBase, both STRING and INTEGER have COMPARABLE as an ancestor, therefore represent proper substitutes for G at run-time.

Listing 2.27: JavaScript Bubble Sort Translation

```
1  var BUBBLE_SORT = runtime.declare("BUBBLE_SORT", [], {
2    sort : function (lst) {
3      var i = 0, sorted = false, temp = null;
4      while (!(sorted)) {
5        sorted = true;
6        i = 1;
7        while (!(i > lst.count())) {
8          if (runtime.special_dispatch(
9              lst.i_th(i-1), "is_greater", lst.i_th(i))) {
10           sorted = false;
11           temp = lst.i_th(i-1);
12           lst.put_i_th(lst.i_th(i), i-1);
13           lst.put_i_th(temp, i);
14         }
15         i = i + 1;
16       }
17     }
18   }
19 });
```

The method `runtime.special_dispatch` has a structure similar to the one presented in Listing 2.28. The method collects the arguments of the call in `args` and then proceeds to test the type of the call target, `obj`, in an outer series of `if`s. Then it tests the feature name, in each case performing the correct operation. If the call target is not a native JavaScript type, then it must be an object of a user-written class and therefore a normal call is made (line 14).

Listing 2.28: `runtime.special_dispatch`

```
1  runtime.special_dispatch = function (obj, feature_name) {
2    var args = Array.prototype.slice.apply(arguments, [2]);
3    if (typeof obj === "number") {
4      if (feature_name === "is_greater") {
5        return obj > args[0];
6      } else if (feature_name === "abs" {
7        ...
8      } else {
9        ...
10     }
11   } else if (typeof obj === "string") {
12     ...
13   } else { ... }
14   return obj[feature_name].apply(obj, args);
15 };
```

It is import to mention that the `runtime.special_dispatch` is generated automatically by the translator from JavaScript EiffelBase equivalents of the classes listed in Section 2.4.3 and therefore needs not be of concern to a programmer.

Chapter 3

Implementation

This chapter describes the implementation of the automatic translator from Eiffel to JavaScript. The translator's source code resides in a project named `javascript_compiler` and depends only on the core of the Eiffel Studio compiler. The class `JAVASCRIPT_COMPILER` is the *facade* [12] for the translator and features `add_class_to_compile` and `execute_compilation` expose it.

The Eiffel compiler implemented and distributed with Eiffel Studio [7] is used in order to parse the Eiffel source code, analyze it semantically and create an abstract syntax tree. Furthermore, the Eiffel compiler resolves types and creates a simplified version of the abstract syntax tree called byte code tree.

Therefore, the translator runs only after the Eiffel compiler has already run and no errors have been found. It is implemented as an Eiffel byte code visitor and translates each byte code node to logically equivalent JavaScript source code, stitching all the different translations together in order to form the translation of complex source code.

The translator features an extensible architecture so that only Eiffel's language features have been implemented in the translator itself. The actual translation rules reside in an external library (see Section 3.2). The mechanism through which they are fed into the translator have been discussed in Section 2.3.

After the desired classes to be translated have been added and feature `execute_compilation` has been called, the translator proceeds to search the system, looking for special translator directives (see Section 2.3.3) and for ancestors of EiffelBase classes which get translated to native JavaScript types (see Section 2.4.4). It populates a `JSC_CLASS_INFORMER` object with the special classes it has found and then proceeds to translate the required classes, one by one.

Listing 3.1: Features from the JSC_CLASS_INFORMER class

```
class
  JSC_CLASS_INFORMER

feature -- Status Report

  is_native_stub (a_class_id: INTEGER): BOOLEAN
    -- Is the class a stub for a native JavaScript object?

  is_fictive_stub (a_class_id: INTEGER): BOOLEAN
    -- Is the class a placeholder of externals?

  is_stub (a_class_id: INTEGER): BOOLEAN
    -- Is the class a stub?

  is_ancestor_of_native_type (a_class_id: INTEGER): BOOLEAN
    -- Is the class an ancestor of an EiffelBase type which
       gets translated to a JavaScript primitive type?

feature -- Basic Operation

  get_native_stub (a_class_id: INTEGER): STRING
    -- Get the qualified name of the native JavaScript class
       for a native stub

  find_class_named (a_class_name: STRING): detachable CLASS_C
    -- Find `a_class_name' in universe.

  is_eiffel_base_class (a_class: CLASS_C): BOOLEAN
    -- Is `a_class` an EiffelBase class?

  redirect_class (a_class: CLASS_C): CLASS_C
    -- Redirect `a_class' to equivalent JavaScript class if `
       a_class' belongs to EiffelBase

  redirect_feature (a_class: CLASS_C; a_feature: FEATURE_I):
     FEATURE_I
    -- Redirect `a_class'.`a_feature' to equivalent JavaScript
        class.feature if `a_class' belongs to EiffelBase.

end
```

Listing 3.1 shows some of the feature of the class JSC_CLASS_INFORMER. This class is used throughout the translation process, as it contains features providing different pieces of information. Feature `redirect_feature` implements the function redirecting EiffelBase calls, μ(C,f) (see Section 2.4.2).

The translation is done through a series of writer classes, each with a

precise scope and each handling a specific part of the translation. The classes work with a byte code tree generated by the Eiffel Studio compiler and are implemented as visitors [12].

- JSC_CLASS_WRITER: Translates an entire Eiffel class to JavaScript, by dispatching to various other writers. It handles the generation of class declarations and invariants.
- JSC_ATTRIBUTE_WRITER: Translates an attribute.
- JSC_DEFAULT_VALUE_WRITER: Handles the default values for different types: **false** for booleans, `0` for numeric types and `null` for objects.
- JSC_CONSTANT_WRITER: Handles the translation of constant literals.
- JSC_SIGNATURE_WRITER: Handles the signature of the JavaScript method representing an Eiffel procedure or a **once** feature.
- JSC_BODY_WRITER: Translates the body of a feature, dispatching the translation of each instruction to JSC_INSTRUCTION_WRITER. It handles the generation of precondition and postcondition checks, the local variables and the equivalent JavaScript code for **rescue** clauses.
- JSC_INSTRUCTION_WRITER: Translates an Eiffel instruction.
- JSC_EXPRESSION_WRITER: Translates an Eiffel expression.

All the writer classes use the JSC_SMART_BUFFER class which is a stack of buffers. This proved to be a very valuable addition to usual buffers, as it often happens that constructs which are nested need to be translated and it is convenient to reuse the same class instances. The generated strings, abstracted by JSC_BUFFER_DATA are "stitched" together, producing the JavaScript output.

Another class used by the writers is JSC_CONTEXT, which provides a shared context for the writers. For example, when translating an instruction, the context contains information such as: the current class and the current feature being translated, the names of the feature's arguments or the names of the feature's local variables, etc. It also provides helper features to create warnings and errors: `add_error` and `add_warning`.

The warnings and errors are stored in SHARED_JSC_ENVIRONMENT, from where they get added to EVE's Error List if the translator is invoked from the Graphical User Interface or they get printed to the standard error output if it is invoked from console.

3.1 EVE integration

The translator is integrated in the Eiffel Verification Environment [9], an experimental branch of Eiffel Studio. From the Graphical User Interface, it is available as a tool and invoking the translator will translate all the classes in the current project to JavaScript (see Fig. 3.1). The tool automatically translates any other referenced classes.

During the translation, the status bar of the tool is updated to reflect the progress and if warnings or errors are found, they are displayed in the Error List (see Fig. 3.2).

Invoking the JavaScript translator from the console is straightforward and requires that the extra flag, `-js_compile`, is used:

```
ec.exe -config prj.ecf -target -js_compile
```

The generated JavaScript files are written to the hard disk in the `EIFGENs/target/JavaScript` folder.

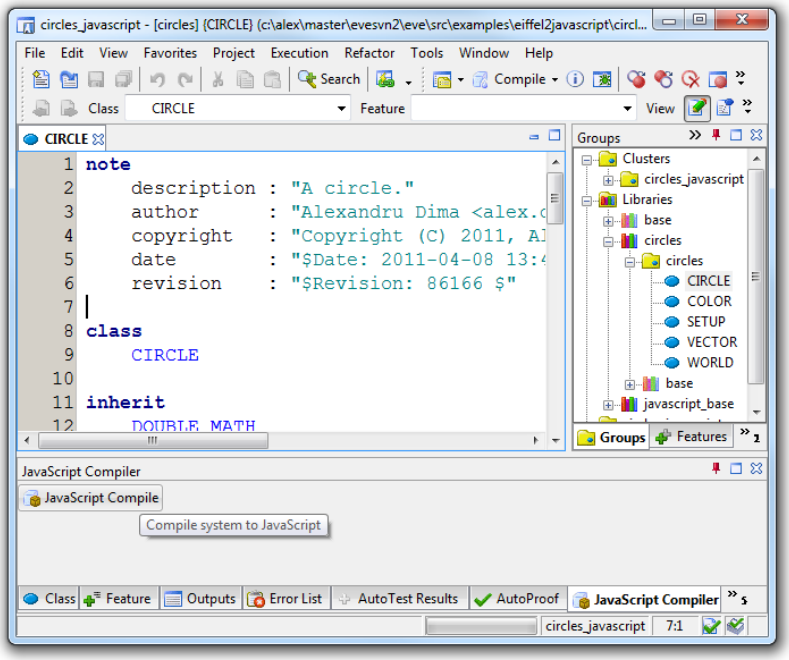

Fig. 3.1: Invoking the translator over the current project

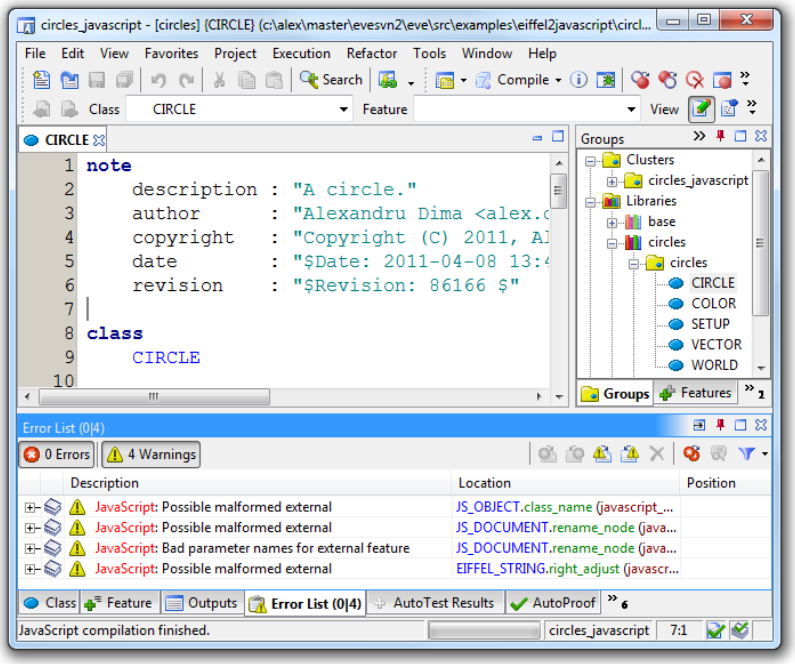

Fig. 3.2: Warnings generated by the translator

3.2 The JavaScript Base library

Because a lot of information has been externalized from the translator, all projects which are translated to JavaScript must include the `javascript_base` library. This library contains essential information the translator needs, such as JavaScript equivalents for EiffelBase classes, but also includes a large collection of stubs for native browser objects.

Using the methodology described in Section 2.3, the library defines *stubs* for native browser objects as specified in:

- *W3C Document Object Model Core Specification* [41]
- *W3C Document Object Model Events Specification* [42]
- *W3C HTML5 Specification* [14]
- *W3C Cascading Style Sheets Object Model Specification* [39]
- *W3C Cascading Style Sheets Object Model View Module Specification* [40]

Te specifications describe the interfaces for the native browser objects in an Interface Definition Language (IDL). Therefore, a parser has been written which reads the interfaces and generates the 126 browser native stubs

included in the `javascript_base`, accounting for more than 10,000 lines of code.

The programmer does not refer directly to the JavaScript equivalents for EiffelBase classes, she writes her code on top of EiffelBase. Through the mechanisms described in Section 2.4 the translator substitutes the used classes with equivalents in the generated source code, transparently to the user. The library contains JavaScript equivalents for 44 EiffelBase classes.

Besides the ones which get translated to native JavaScript types, as presented in Section 2.4.3, the library also contains JavaScript equivalents for the following EiffelBase classes:

- `LIST`, `ARRAYED_LIST`, `LINKED_LIST`
- `ARRAY2`
- `HASH_TABLE`
- `SET`, `LINKED_SET`
- `TREE`, `LINKED_TREE`

These classes provide a minimum set of data structures an application needs. They are supported because they have been needed during the development of the case studies (see Chapter 4).

3.3 Testing

Tests have been written in order to prevent breaking changes in the translator. They ensure that the translator functions end-to-end: the tests are written in Eiffel, they get translated to JavaScript and are executed inside a browser. Besides serving as smoke-tests, they validate two aspects of the translation:

- the fact that Eiffel language features are translated in such a way that their original behaviour is the same in the translated code.
- the fact that the EiffelBase equivalents are implemented correctly.

To support these tests, a helper class, named `TEST`, has been created which has helper features such as: `invoke_test`, `assert` or `expects_exception` to ease the writing of tests.

Listing 3.2: Testing object tests

```
class
  TEST_OBJECT_TESTS
inherit
  TEST

feature {NONE} -- Initialization
  make
    do
      invoke_test ("generics", agent
        local
          l: LIST[STRING]
        do
          create {LINKED_LIST[attached STRING]}l.make

          assert (attached {LINKED_LIST[attached STRING]} l)
          assert (attached {LINKED_LIST[STRING]} l)
          assert (attached {LIST[attached STRING]} l)
          assert (attached {LIST[attached ANY]} l)
          assert (attached {LIST[ANY]} l)
          assert ({LIST[STRING]} l)
          assert (not attached{LIST[INTEGER]} l)
        end
      )
    end
end
```

Listing 3.2 shows a test which validates an Eiffel language feature: that object tests function correctly with generic types. Listing 3.3 shows a test which validates that the equivalent for EiffelBase class `LINKED_SET` works correctly in a border case.

Listing 3.3: Testing the EiffelBase equivalent for `LINKED_SET`

```
class
  TEST_LINKED_SET
inherit
  TEST
feature {NONE} -- Initialization
  make
    do
      invoke_test ("one_element_set_test", agent
        local
          l_set: SET[attached STRING]
        do
          create {LINKED_SET[attached STRING]}l_set.make
          assert (l_set.is_empty)

```

```
15            l_set.extend ("foo")
16            assert (not l_set.is_empty)
17            assert (l_set.has ("foo"))
18            assert (l_set.count = 1)
19
20            l_set.prune ("foo")
21            assert (l_set.is_empty)
22            assert (not l_set.has ("foo"))
23            assert (l_set.count = 0)
24          end
25        )
26      end
27  end
```

The test suite has been run before every commit of a translator change to the configuration management tool used for development (a SVN repository), effectively reducing the number of introduced defects.

3.4 Limitations

This section lists the unsupported Eiffel language features and other limitations of the current implementation of the translator. These issues do not reflect problems in the general approach, but rather refer to the specific current implementation:

- not supported: renaming inherited attributes
- not supported: feature selection in multiple inheritance
- not supported: the `default_rescue` feature.
- semantic difference: strings in Eiffel are mutable, while in JavaScript they are immutable.
- constants are inlined
- a class with multiple generics does not name what generic it uses when inheriting from a class with only one generic.

Some solutions to these issues are described in Section 6.2.

Chapter 4

Case Study

This chapter presents some of the applications written in Eiffel and translated to JavaScript using the automatic translator. These applications have been developed mostly in order to test the translator, but to also show different use cases for it. The circles example shows how code can be reused and the editor example is a full-fledged web-based application, proving the applicability of the translator for a large project.

4.1 Circles

The first example developed as a show case for the translator aims to prove one of the translator's goals: code reusability. Two applications have been created: one which runs natively, with an user interface built with EiffelVision2 [8] and another one which runs in a browser, with an user interface built with JavaScript Base (Section 3.2).

The example consists of a world (a fixed size rectangle) filled with moving circles of different colors and sizes. A rudimentary physics engine has been written to simulate totally elastic collision between circles. The formulas to compute the velocities of two colliding objects have been found on Wikipedia [23]:

$$v_1 = \frac{m_1 - m_2}{m_1 + m_2} u_1 + \frac{2m_2}{m_1 + m_2} u_2$$

$$v_2 = \frac{m_2 - m_1}{m_1 + m_2} u_2 + \frac{2m_1}{m_1 + m_2} u_1$$

, where v_1 and v_2 are the final velocities of the circles, u_1 and u_2 are the initial velocities and m_1 and m_2 are the masses.

The classes which model the world, the circles and their collision are independent to the user interface. Therefore, they have been grouped into a separate project (a library):

- VECTOR: a 2D vector with common operations like addition, subtraction, dot product, normalization, scalar multiplication, scalar division, etc.
- COLOR: representation for color: both in html form and in RGB (red, green, blue).
- CIRCLE: a circle with a position, radius, mass, color and velocity (see Listing 4.1).
- WORLD: a rectangle containing circles. Has a feature `update` which updates the state of the circles with `a_delta_time` milliseconds.
- SETUP: a class which creates a world with 7 circles of different colors and sizes.

Listing 4.1: A class modelling a circle

```
1  class
2    CIRCLE
3
4  inherit
5    DOUBLE_MATH
6
7  create
8    make
9
10 feature {NONE} -- Initialization
11
12   make (a_pos: VECTOR; a_radius: DOUBLE; a_color: COLOR)
13     do
14       ...
15     end
16
17 feature -- Access
18
19   color: COLOR
20   radius, mass: DOUBLE
21   pos: VECTOR
22   velocity: VECTOR
23
24 end
```

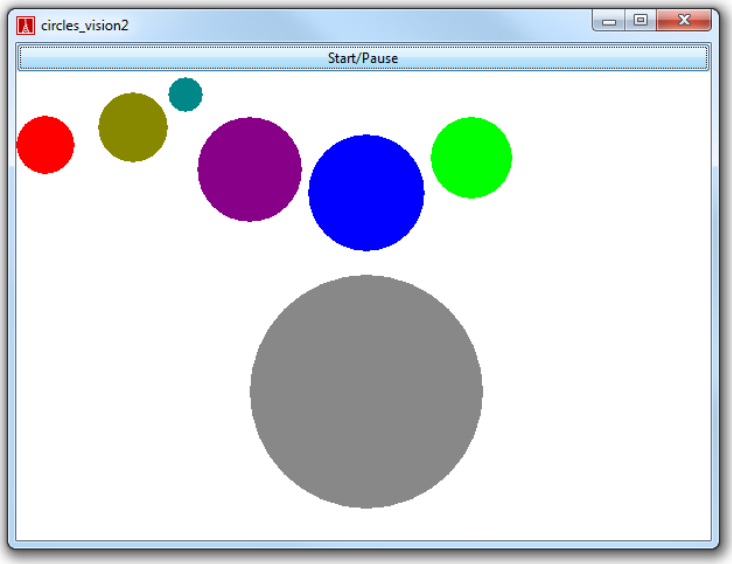

Fig. 4.1: The circles example running as a native application

The library containing the model classes is used by an EiffelVision2 application, which contains a simple window with a background thread that updates and renders the updated world. Listing 4.2 shows how the circles are rendered using EiffelVision2. A pixel map is created and each circle is drawn onto it. Later on, the pixel map is drawn on the window. Figure 4.1 shows a still screenshot of running the EiffelVision2 application.

Listing 4.2: Rendering the circles in EiffelVision2

```
1  render: attached EV_PIXMAP
2    local
3      color: EV_COLOR
4      c: CIRCLE
5    do
6      create Result.make_with_size (world.width, world.height)
7      from world.circles.start until world.circles.after loop
8        c := world.circles.item
9
10       create color.make(c.color.r, c.color.g, c.color.b)
11       Result.set_foreground_color (color)
12       Result.fill_ellipse (c.pos.x-c.radius, c.pos.y-c.radius,
             2*c.radius, 2*c.radius)
13
14       world.circles.forth
15     end
16   end
```

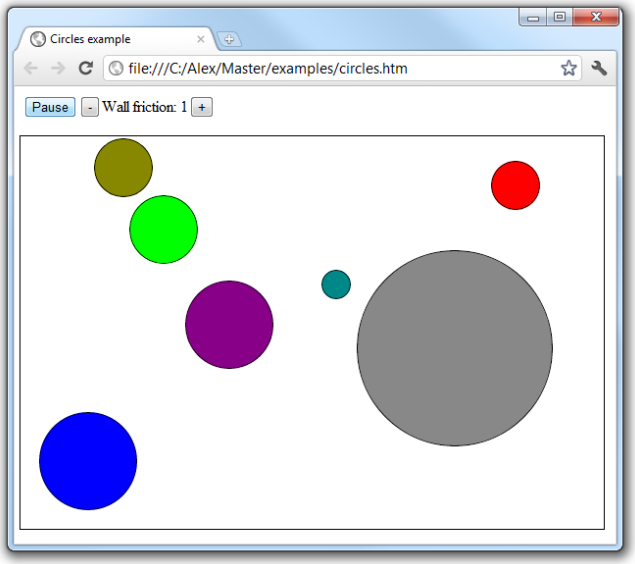

Fig. 4.2: The circles example running in a browser

The same library containing the model classes is also used by a second application, one which contains a JavaScript user interface. Listing 4.3 shows how the circles are rendered using `div` DOM nodes. Figure 4.2 shows a still screenshot of running the application in a browser.

Listing 4.3: Rendering the circles in the browser

```
1  render
2    local
3      dom_node: JS_HTML_DIV_ELEMENT; c: CIRCLE
4    do
5      from world.circles.start until world.circles.after loop
6        c := world.circles.item
7
8        create dom_node.make
9        dom_node.style.position := "absolute"
10       dom_node.style.left := (c.pos.x - c.radius).out + "px"
11       dom_node.style.top := (c.pos.y - c.radius).out + "px"
12       dom_node.style.width := (2 * c.radius).out + "px"
13       dom_node.style.height := (2 * c.radius).out + "px"
14       dom_node.style.background := c.color.html
15       set_border_radius (dom_node, c.radius.out + "px")
16
17       world_dom_node.append_child (dom_node)
18       world.circles.forth
19     end
20   end
```

In conclusion, this example shows how the same Eiffel source code is both compiled to C, as part of a native application and how it is translated to JavaScript, as part of an application running in the browser.

4.2 The editor

As part of this thesis, a full-fledged web-based source code editor has been developed. In its incipient form, the editor served for testing the translator, but soon took off to become a powerful standalone application. In fact, the editor influenced greatly the development of the translator itself, as it became the main driving force behind translator features. For example, the supported EiffelBase classes (see Section 3.2) came from the needs encountered during the development of the editor.

The main features of the editor are:

- source code highlighting
- concurrent editing by multiple developers with an implicit line-based versioning system
- virtual rendering (only the visible lines are actually rendered)
- commenting/uncommenting multiple lines of code
- indenting/unindenting multiple lines of code
- full mouse support
- copy-pasting
- searching in file
- support for annotating lines
- handling of variable width fonts
- handling of thousands of lines of code

As a full description of the editor is outside the scope of this document, this section only provides a brief explanation of how the editor works.

The motivation for the editor appeared when several problems were identified in the previous editor used in the CloudStudio IDE: there was no separation between the view and the model, it was implemented with a browser

`textarea`, thus prohibiting any sort of extensions; it offered no built-in support for concurrent editing or for restoring the cursor position.

The new editor is implemented with a clear separation between the view and the model and the communication between them is done exclusively with events. The model in this case consists of a list of lines, each with a proper unique identifier and with the capability to represent multiple versions of its content. View options are then applied to the model, basically acting as a filter and choosing which version of a line is rendered on screen.

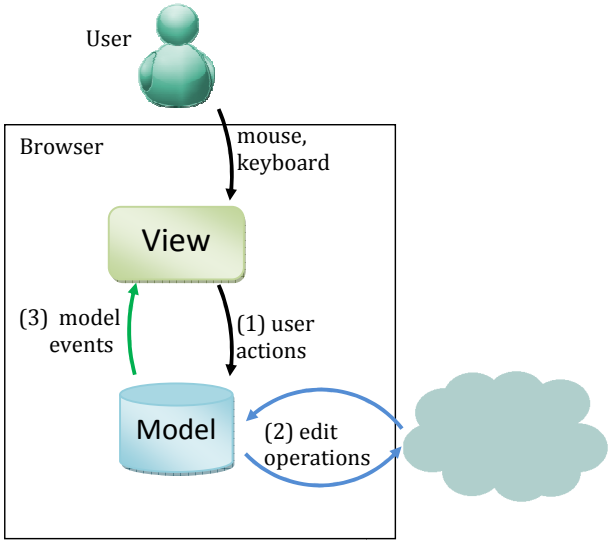

Fig. 4.3: Events in the editor

Since the view is completely separated from the model and it updates on model events, the following two scenarios are possible:

- **From the view to the model**. The user can trigger an action by using the mouse, the keyboard (e.g. clicking, dragging, typing, pasting, etc.) or by clicking on buttons. When a user action has been triggered, an event describing it is generated ((1) in Fig. 4.3):
  - move cursor
  - press the delete/backspace keys
  - press the enter key
  - press other keys
  - cut

  - paste
  - select all
  - commit
  - rollback

  These events reach the class modeling the cursor. Based on the current cursor position and on the current selection, the cursor *transforms* the events in one or more of the following logical operations ((2) in Fig. 4.3):

  - insert new lines
  - change existing lines
  - delete lines
  - commit lines
  - rollback lines

  These operations are then relayed to the model **and to the server**.

- **From the model to the view**. When the current user has triggered a series of line operations or when such operations are received from the server, the model reacts and properly treats them (e.g. new lines are added to the model or new versions for a line's content are created, etc.). After acting on the operations, a further series of events are broadcasted to the view ((3) in Fig. 4.3)::

  - cursor change
  - line change
  - line annotation change
  - delete lines
  - insert lines
  - scroll view

  The latter series of events are processed by the view, which, given the current visible lines, updates the DOM nodes representing the source code accordingly.

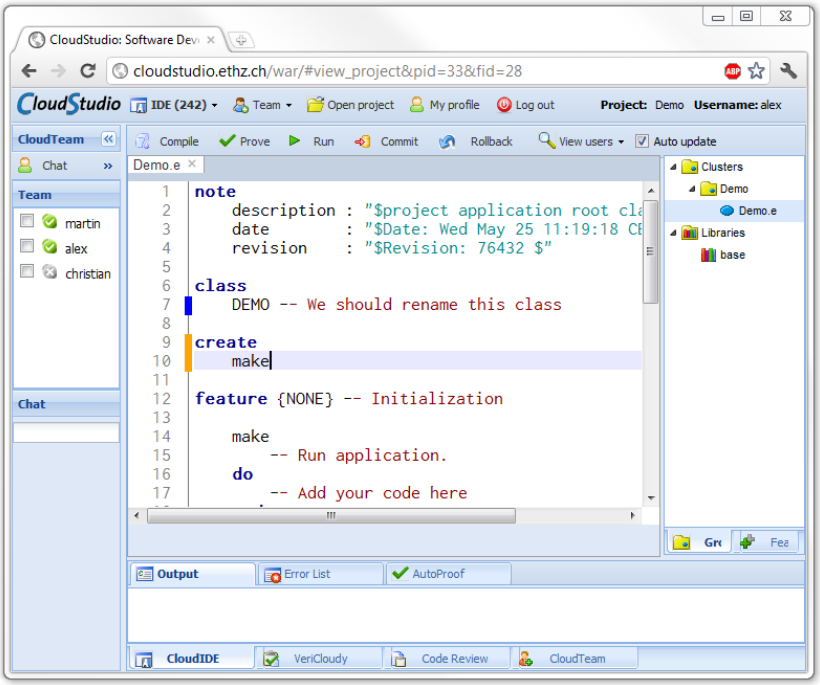

Fig. 4.4: Viewing other users changes

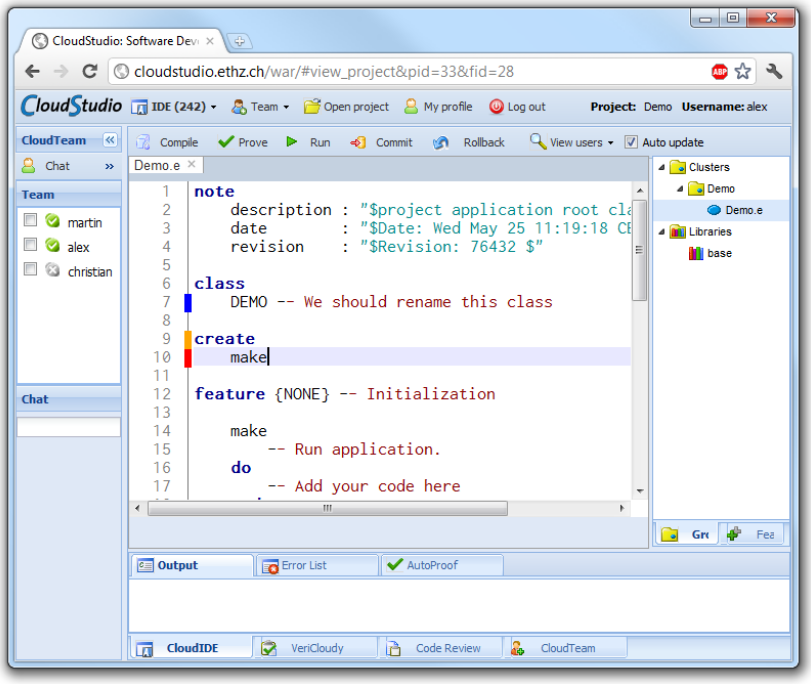

Fig. 4.5: A conflict is seen in real-time

Chapter 5

Related Work

5.1 JavaScript Language Translators

Google Web Toolkit (GWT) [13] is an open source framework for creating web-based applications, with a Java to JavaScript compiler at its heart. The main differences are that the GWT compiler produces object-oriented bytecode, the JavaScript object model closely resembles Java's (i.e. the model supports single inheritance) and it has a monolithic approach to compilation, in order to optimize the generated code. The translator developed in this thesis produces JavaScript code which is human-readable and actually looks like source code that a programmer could have written. This allows to take full advantage of the tooling available in browsers (e.g. for debugging, for performance testing, etc.). Also, the translator allows for dynamic (i.e. at run-time) definition of new classes.

JDojo [16] is a Java to JavaScript compiler integrated in the Eclipse Java Compiler [17] which generates human-readable JavaScript code. The main differences are that JDojo creates JavaScript source code which only works with the Dojo toolkit [35] and that "the programmer does not program against the Java JDK classes, but against Dojo and JavaScript stubs that JDojo provides" [16]. The translator developed in this thesis generates JavaScript source code which runs independent on the existence of a library and allows programmers to write code against EiffelBase.

5.2 Eiffel Language Translators

AutoProof [24, 37] is a translator which transforms Eiffel programs to an intermediate language, BoogiePL [6], and then uses the Boogie verifier to

check if the program satisfies the specifications. This compiler only covers a subset of Eiffel and it focuses on developing techniques to verify programs. This thesis develops a translator from Eiffel to JavaScript which covers a much higher percentage of Eiffel's features.

Trudel et al. [36] implemented a compiler that takes Java programs and produces Eiffel programs. The translation is done from a high level language (Java) to another high level language (Eiffel), similar to the translator of this thesis which also transforms a high level language (Eiffel) to another high level language (JavaScript). The main difference is that in Trudel et al.'s work [36], the input object model is similar to the output object model. This is not the case in this work, since a suitable representation for Eiffel classes in JavaScript had to be defined.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An automatic translator that transforms Eiffel programs to JavaScript programs has been designed and implemented. It supports the most important Eiffel language features, including agents, contracts, exception handling, multiple inheritance and once routines, mapping Eiffel concepts to counter-parts in JavaScript or emulating them where no JavaScript equivalents exist.

The translator is built with an extensible architecture and with mechanisms to enhance the translation rules. A library consisting of 126 classes describing native JavaScript objects (i.e. stubs) and of equivalents for 44 EiffelBase classes enhances the translator.

The applicability of the translator has been demonstrated with several case studies. One shows how the same Eiffel source code can be either compiled to C or translated to JavaScript without any changes. The other case study is a full-fledged web-based source code editor. The editor has been integrated to the CloudStudio IDE, making up for 10,000 lines of code or 35% of the project.

6.2 Future work

The work presented in this thesis offers several opportunities for further research and improvement. At first, the limitations identified in Section 3.4 may be addressed. They represent restrictions of the current implementation which can be resolved through additional implementation effort:

- renaming attributes can be achieved by introducing a getter and a setter for each attribute of a class. Attributes should then be accessed only through those methods. When renamed, the two methods may be overwritten by the `runtime` to assign and read the new attribute. However, accessing attributes through methods all the time would lead to a decrease in performance and it might therefore be acceptable that the translator does not support attribute renaming.

- feature selection may be introduced as another parameter in each object of the `parents` array used when declaring a class. Then, the `runtime.declare` can be enhanced to take feature selection into consideration.

- constants may be referred to statically rather than inlined.

- when inheriting from a class which has generics, an extra parameter in each object of the `parents` array may account for what generic is used to inherit from the parent class. The `runtime.inherits` can be enhanced to take this into account.

Besides fixing the limitations in the implementation, the following ideas can be considered future work:

- adding more EiffelBase equivalents to the `javascript_base` library.

- developing a new implementation of EiffelVision2, one which is built on top of `javascript_base`, such that programmers write their User Interface code once against EiffelVision2 and then they can translate their code to JavaScript and run it in the browser.

- writing helper methods to *hide* and provide safe wrappers around common entry points for DOM-based vulnerabilities: around the `innerHTML` property of `HTMLElements`, around `window.location`, etc. Then, a new subclass of `STRING` may be created, `SAFE_STRING` for example, which can be used to achieve statical, compile-time safety.

Bibliography

[1] ECMA-262: ECMAScript Language Specification, December 2009. Available at http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] Adobe acknowledges critical security flaw in software. http://www.bbc.co.uk/news/10257411, May 2011.

[3] Adobe Flash. http://www.adobe.com/software/flash/about/, May 2011.

[4] Applets. http://java.sun.com/applets/, May 2011.

[5] D. Crockford. Prototypal inheritance in javascript. http://javascript.crockford.com/prototypal.html, Apr. 2008.

[6] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[7] Eiffel Software. EiffelStudio. A Complete Integrated Development Environment. http://www.eiffel.com/products/studio/, May 2011.

[8] Eiffel Software. EiffelVision2. A platform independent Graphical User Interface (GUI) library. http://www.eiffel.com/libraries/vision2.html, May 2011.

[9] Eiffel Verification Environment (EVE). http://eve.origo.ethz.ch/.

[10] EiffelBase. http://docs.eiffel.com/book/solutions/eiffelbase, May 2011.

[11] Expert says Adobe Flash policy is risky. http://news.cnet.com/8301-27080_3-10396326-245.html, May 2011.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Google Web Toolkit. `http://code.google.com/webtoolkit/`, May 2011.

[14] Hypertext Markup Language (HTML) Specification 5th major revision. `http://dev.w3.org/html5/spec/Overview.html`, May 2011.

[15] JavaScript:TraceMonkey – MozillaWiki. `https://wiki.mozilla.org/JavaScript:TraceMonkey`, May 2011.

[16] JDojo. `https://jazz.net/wiki/bin/view/Main/JDojo`, May 2011.

[17] JDT Core Component – Eclipse. `http://www.eclipse.org/jdt/core/`, May 2011.

[18] B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[19] B. Meyer. Applying "Design by Contract". *IEEE Software*, 25(10):40–51, 1992.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[21] B. Meyer. The Unspoken Revolution in Software Engineering. *IEEE Computer*, 39(1):121–124, 2006.

[22] B. Meyer (editor). ISO/ECMA Eiffel standard (Standard ECMA-367: Eiffel: Analysis, Design and Programming Language), June 2006. available at `http://www.ecma-international.org/publications/standards/Ecma-367.htm`.

[23] Momentum – Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Momentum&oldid=430943981`, May 2011.

[24] M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In J. Vitek, editor, *TOOLS-EUROPE*, LNCS, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] M. Nordio, H.-C. Estler, B. Meyer, J. Tschannen, C. Ghezzi, and E. D. Nitto. How do Distribution and Time Zones affect Software Development? A Case Study on Communication. In *6th International Conference on Global Software Engineering*. IEE, 2011.

[26] M. Nordio, C. Ghezzi, B. Meyer, E. D. Nitto, G. Tamburrelli, J. Tschannen, N. Aguirre, and V. Kulkarni. Teaching Software Engineering using Globally Distributed Projects: the DOSE course. In *Collaborative Teaching of Globally Distributed Software Development - Community Building Workshop (CTGDSD)*, New York, NY, USA, 2011. ACM.

[27] M. Nordio, M. Joseph, B. Meyer, and A. Terekhov. Software Engineering Approaches for Outsourced and Offshore Development, 4th International Conference, St. Petersburg, Russia. Lecture Notes in Business Information Processing 54, Springer-Verlag, 2010.

[28] M. Nordio, B. Meyer, and H.-C. Estler. Collaborative Software Development on the Web, 2011.

[29] M. Nordio, R. Mitin, and B. Meyer. Advanced Hands-on Training for Distributed and Outsourced Software Engineering. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, New York, NY, USA, 2010. ACM.

[30] M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. D. Nitto, and G. Tamburelli. The Role of Contracts in Distributed Development. In *Software Engineering Approaches for Offshore and Outsourced Development*, volume 35 of *LNBIP*, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] Pwn2Own 2010: interview with Charlie Miller. `http://www.oneitsecurity.it/01/03/2010/interview-with-charlie-miller-pwn2own/`, May 2011.

[32] Rich Internet Application Market Share. `http://www.statowl.com/custom_ria_market_penetration.php`, May 2011.

[33] Rich Internet Application Statistics. `http://riastats.com/`, May 2011.

[34] Silverlight. `http://www.microsoft.com/silverlight/`, May 2011.

[35] The Dojo Toolkit. `http://dojotoolkit.org/`, May 2011.

[36] M. Trudel, M. Oriol, C. A. Furia, and M. Nordio. Automated Translation of Java Source Code to Eiffel. In *TOOLS-EUROPE*, LNCS, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] J. Tschannen. Automatic Verification of Eiffel Programs. Master's thesis, ETH Zurich, 2009.

[38] V8 JavaScript Engine. http://code.google.com/p/v8/, May 2011.

[39] World Wide Web Consortium Cascading Style Sheets Object Model. http://dev.w3.org/csswg/cssom/, May 2011.

[40] World Wide Web Consortium Cascading Style Sheets Object Model View Module. http://www.w3.org/TR/cssom-view/, May 2011.

[41] World Wide Web Consortium Document Object Model Core. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html, May 2011.

[42] World Wide Web Consortium Document Object Model Events. http://dev.w3.org/2006/webapi/DOM-Level-3-Events/html/DOM3-Events.html, May 2011.