Using JavaScript as a Real Programming Language

Tommi Mikkonen and Antero Taivalsaari

Using JavaScript as a Real Programming Language

Tommi Mikkonen and Antero Taivalsaari

SMLI TR-2007-168 October 2007

Abstract:

With the increasing popularity of the World Wide Web, scripting languages and other dynamic languages are currently experiencing a renaissance. A whole new generation of programmers are growing up with languages such as JavaScript, Perl, PHP, Python and Ruby. The attention that dynamic languages are receiving is remarkable, and is something that has not occurred since the early days of personal computers and the BASIC programming language in the late 1970s and early 1980s.

At the same time, the web is becoming the *de facto* target platform for advanced software applications, including social networking systems, games, productivity applications, and so on. Software systems that were conventionally written using static programming languages such as C, C++ or Java™, are now built with dynamic languages that were originally designed for scripting rather than full-scale application development.

At Sun Labs, we have created a new, highly dynamic web programming environment called the *Lively Kernel* that is built entirely around JavaScript. As part of this effort, we have written a lot of JavaScript code and applications that exercise the JavaScript language in a different fashion than the typical JavaScript programs found on commercial web sites. Among other things, we have used JavaScript as a systems programming language to write the Lively Kernel itself.

In this paper we summarize our experiences using JavaScript, focusing especially on its use as a real, general-purpose programming language.

Sun microsystems

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
tommi.mikkonen@sun.com
antero.taivalsaari@sun.com

Using JavaScript as a Real Programming Language

Tommi Mikkonen   Antero Taivalsaari
tommi.mikkonen@sun.com   antero.taivalsaari@sun.com

Sun Microsystems Laboratories
P.O. Box 553 (TUT)
FIN-33101 Tampere, Finland

1. Introduction

With the increasing popularity of the World Wide Web, scripting languages and other dynamic languages are currently experiencing something of a renaissance. A whole new generation of programmers are growing up with languages such as JavaScript, Perl, PHP, Python and Ruby. The attention that dynamic languages are receiving is remarkable, and is something that has not occurred since the early days of personal computers and the widespread use of the BASIC programming language in the late 1970s and early 1980s.

At the same time, the web is becoming the *de facto* target platform for advanced software applications, including social networking systems, games, productivity applications, and so on. Software systems and applications that were conventionally written using a static programming language such as C, C++ or Java™, are now built with dynamic languages that were originally designed for scripting rather than full-scale, general-purpose application development. Such systems and applications are targeted to the web, to be used by a large number of users from their web browsers, rather than to any specific operating system, computer or device. Examples of technologies supporting web application development include Ajax [CPJ05], Google Web Toolkit [Pra07] and Ruby on Rails [Tat06], to name a few.

At Sun Labs, we have created a new, highly dynamic web programming environment called the *Lively Kernel* (or *Sun Labs Lively Kernel*) that is built entirely around the JavaScript programming language. As part of this effort, we have written a lot of JavaScript code and applications that exercise JavaScript language features in a different fashion than the typical JavaScript programs found on commercial web sites. Among other things, we have used JavaScript as a systems programming language to write the Lively Kernel system itself, much in the same way that integrated development environments and other system-level facilities were written earlier for systems such as Smalltalk [GoR83] and Self [UnS87]. This is in striking contrast to the conventional use of the JavaScript language; on the web today, JavaScript is used primarily for decorating and animating web pages, or for performing some other relatively limited scripting tasks.

This paper summarizes our experiences in using JavaScript, focusing especially on its use as a general-purpose programming language, as opposed to its conventional use as a scripting language. The paper is structured as follows. Section 2 provides a brief introduction to dynamic languages and JavaScript. Section 3 provides an overview of the Lively Kernel, including a summary of the underlying technology and the types of applications that we have developed. Section 4 summarizes our experiences with JavaScript. Finally, Section 5 concludes with what has been accomplished.

2. Dynamic Languages and JavaScript

2.1 Dynamic Languages

The term *dynamic programming language* describes a class of programming languages that share a number of common runtime characteristics that are available in static languages only during compilation, if at all. These behaviors can include the ability to extend the currently running program by loading or generating new code by extending the classes, objects and other definitions of the program, or even by modifying the internals of the language itself, all during program execution. While these behaviors can be emulated in almost any language of sufficient complexity, such behaviors are integral, built-in characteristics of dynamic languages.

A dynamic language possesses one or more of the following characteristics:

1) *Dynamic typing*. In static languages such as C, C++ and Java, variable and parameter types must be explicitly assigned before compilation. With most dynamic languages, however, variables need not be declared before their use. Furthermore, their type is not determined until runtime when the type information is actually needed.

2) *Interpretation.* With static languages, code is compiled into a binary representation or some other intermediate form before execution. With dynamic languages, source code is read at runtime, translated into an intermediate representation or machine code dynamically and then executed immediately. From the viewpoint of the end user, all these phases occur seamlessly and automatically.

3) *Runtime modification*. When using static programming languages, code structures are immutable at runtime apart from some limited extensibility, for instance, in the form of dynamically linked libraries (DLLs) or other plug-in components. With dynamic languages, however, class hierarchies and other structural and behavioral aspects of the program can be modified at runtime. For instance, new functions and variables can be added to classes and objects on the fly.

Because of their flexibility and malleability, dynamic programming languages are often used for "exploratory" programming – a programming style in which programs are developed in an evolutionary fashion, often utilizing integrated development tools that are part of the system itself. Examples of such languages and systems include Smalltalk and Self. These languages include an integrated development and debugging environment that allows the programmers to "live" inside the system itself. In such a system, the distinction between the application, language and the development environment can be blurry. This has been a point of contention to many critics as it makes it difficult to deploy applications separately from the development environment.

With the growing popularity of the World Wide Web, dynamic languages are experiencing a comeback. The most commonly used dynamic languages on the web include JavaScript, Perl, PHP, Python and Ruby. These languages are commonly referred to as *scripting* languages, although the languages themselves place no specific limitations on their use for purposes other than scripting.

2.2 JavaScript

JavaScript [Fla06] is an object-oriented programming language based on the prototype-based object model [NTM99]. The language is best known for its use as a scripting language on the web. JavaScript is a key building block of *Dynamic HTML* (DHTML) [Goo06]: a collection of technologies that are included in nearly all web browsers to support the creation of animated and interactive web sites. When integrated inside the web browser, the JavaScript implementation includes a set of libraries that are collectively referred to as "*Client-Side JavaScript.*" In contrast, the JavaScript language and the core JavaScript libraries (that is, those libraries that are independent of the web) are usually referred to as "*Core JavaScript.*" The comments presented in this paper pertain mainly to Core JavaScript.

JavaScript is a typical dynamic language in the sense that variables in JavaScript do not need to be introduced before their use and that the types of variables are resolved dynamically during execution. JavaScript allows function definitions and other code to be modified while the program is running. The execution model of JavaScript is based on interpretation of source code. In contrast with less dynamic languages such as the Java programming language, there is no public intermediate representation format such as class files or binary files. In summary, JavaScript manifests all the three key characteristics of dynamic languages listed in Section 2.1.

Despite its name, JavaScript is only a distant relative of the Java programming language. The main commonality between the languages lies in the syntactic similarity that both Java and JavaScript share with the C programming language. Semantically, JavaScript is much closer to dynamic programming languages such as Smalltalk, Self, or even Lisp.

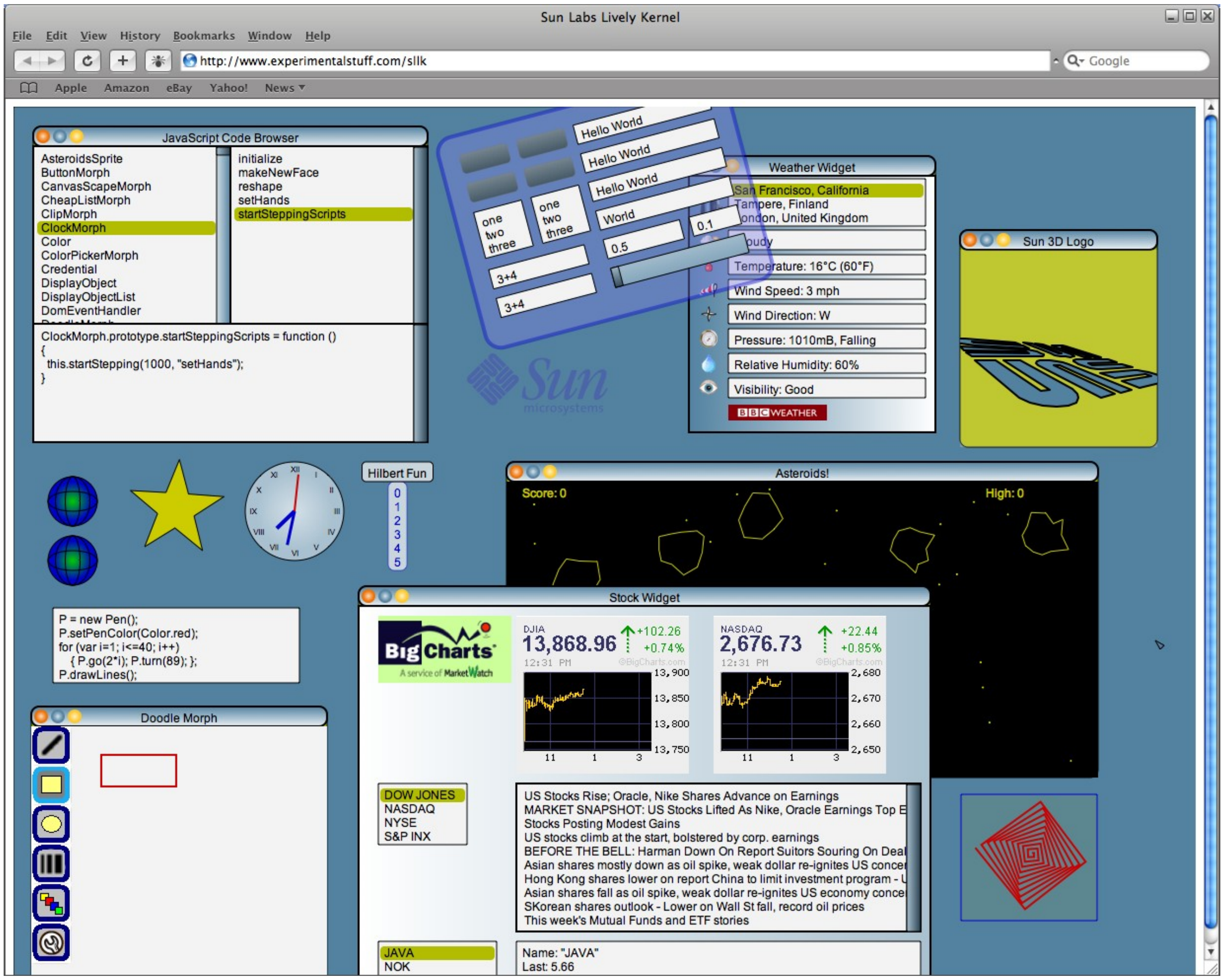
JavaScript has suffered from its reputation as a "toy" language that is useful only for relatively simple scripting tasks. Even though JavaScript is a general-purpose programming language, so far its use has been limited primarily to web content scripting and animation. However, with the introduction of Web 2.0 technologies such as Ajax, the average size of JavaScript applications is growing. As more and more advanced JavaScript applications and libraries are created, the negative perception about JavaScript will gradually improve.

3. The Lively Kernel – An Overview

3.1 Introduction to the Lively Kernel

The Lively Kernel is a new, highly dynamic web programming environment built around JavaScript. The key idea of the Lively Kernel is to support web applications that provide a user experience and direct manipulation capabilities that are at least as good as those in the best desktop applications, but without the installation and upgrade hassles that conventional desktop applications have. The Lively Kernel is intended to run in any commercial web browser without any installation or plug-in components whatsoever.

A key difference between the Lively Kernel and other projects in the same area is our focus on *uniformity*. Our goal is to build a platform using a minimum number of underlying technologies. This is in contrast with many current web technologies that utilize a diverse array of technologies such as HTML, CSS, DOM, JavaScript, PHP, XML, and so on. In the Lively Kernel we attempt to do as much as possible using a single technology: JavaScript. We have chosen JavaScript primarily because of its ubiquitous availability in the web browsers today and because of its syntactic similarity to other highly popular languages such as C, C++ and Java. However, we also want to leverage the dynamic aspects of JavaScript, especially the ability to modify applications at runtime. Such capabilities are an essential ingredient in building a malleable web programming environment that allows applications to be developed interactively and collaboratively.



```
ClockMorph.prototype.startSteppingScripts = function ()
{
 this.startStepping(1000, "setHands");
}
```

San Francisco, California
Tampere, Finland
London, United Kingdom
Cloudy
Temperature: 16°C (60°F)
Wind Speed: 3 mph
Wind Direction: W
Pressure: 1010mB, Falling
Relative Humidity: 60%
Visibility: Good

```
P = new Pen();
P.setPenColor(Color.red);
for (var i=1; i<=40; i++)
    { P.go(2*i); P.turn(89); };
P.drawLines();
```

DJIA 13,868.96 +102.26 +0.74% 12:31 PM
NASDAQ 2,676.73 +22.44 +0.85% 12:31 PM

DOW JONES
NASDAQ
NYSE
S&P INX

US Stocks Rise; Oracle, Nike Shares Advance on Earnings
MARKET SNAPSHOT: US Stocks Lifted As Nike, Oracle Earnings Top E
Stocks Posting Modest Gains
US stocks climb at the start, bolstered by corp. earnings
BEFORE THE BELL: Harman Down On Report Suitors Souring On Deal
Asian shares mostly down as oil spike, weak dollar re-ignites US concer
Hong Kong shares lower on report China to limit investment program - U
Asian shares fall as oil spike, weak dollar re-ignites US economy conce
SKorean shares outlook - Lower on Wall St fall, record oil prices
This week's Mutual Funds and ETF stories

JAVA
NOK

Name: "JAVA"
Last: 5.66

Figure 1. The Lively Kernel system running a number of widgets and applications

A key component of the Lively Kernel is a graphics framework called *Morphic*[1]. Morphic is a user interface framework that supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as displays, fonts and color maps. A primary goal of Morphic is to make it easy to construct and edit interactive graphical objects, both by direct manipulation and from within programs. The Morphic user interface was originally developed for the Self system [Mal95, MaS95], but it became popular later also as part of the Squeak system [IKM97].

The Lively Kernel system currently consists of about ten thousand lines of uncompressed, unobfuscated

1 The JavaScript implementation of Morphic has been written by other members of our project team. The system will be described in more detail in a separate paper.

JavaScript code, including the Morphic graphics framework described above. Figure 1 provides a screen snapshot of the Lively Kernel, including a number of widgets, applications and tools.

To test the Lively Kernel system yourself, go to the following URL:

http://research.sun.com/projects/lively

The Lively Kernel is interesting in the sense that it demonstrates the ability to run a self-sufficient, visual JavaScript development environment on top of a small kernel that has been written entirely in the JavaScript language. By *self-sufficient* we mean that it is possible to "live inside" the Morphic worlds (the equivalent of workspaces in some other systems) and perform all the necessary object creation and manipulation operations using a visual, fully interactive environment that has been built entirely in JavaScript. As such, the Lively Kernel is an ideal research vehicle for studying the capabilities of JavaScript as a general-purpose programming language as well as a systems programming language.

3.2 Applications

In order to demonstrate the capabilities of the Lively Kernel, we have written or ported a number of applications entirely in JavaScript. The size of the individual applications currently ranges from a few hundred lines to about fifteen thousand lines. The types of applications are listed in Table 1.

Application type	*Examples*
Games	A number of games, including Asteroids, a car racing game, a collaborative version of Pac Man, and a suite of smaller games.
Information applications	An RSS feed reader, stock widget, weather widget, and a number of map applications, including a collaborative application that allows the users to share and annotate maps over the web.
Productivity applications	A simple instant messenger program, a collaborative visual personal organizer program, and a collaborative spreadsheet application that allows the spreadsheet data to be shared and edited over the web.
Programming and debugging tools	A visual JavaScript class browser and object inspector that allow JavaScript code to be edited interactively, as well as a style panel for editing the visual characteristics of objects; in addition, we have developed some tools for debugging and profiling JavaScript code.

Table 1. Types of applications developed for the Lively Kernel

4. Experiences with JavaScript

In this section we summarize our experiences using JavaScript as a real programming language. Our experiences are based on JavaScript language version 1.5, which is also often referred to as ECMAScript version 3 [ECM99]. There are later versions of the JavaScript language, but the newer versions are not yet as popular as version 1.5. JavaScript language version 2.0 has been under specification for several years, but the proposed changes have been so controversial that JavaScript 2.0 has not received much support from the industry.

We have divided our comments on JavaScript into the following categories:

– language comments,
– library comments,
– software configuration comments,
– development style comments,
– virtual machine comments, and
– other comments and observations.

4.1 Language comments

Below we provide some general comments on the JavaScript language itself. We have intentionally kept these comments at a fairly high level, avoiding detailed syntactic and semantic issues. Such issues would be better covered in another, more comprehensive paper.

Extreme permissiveness. JavaScript is an extremely permissive, error-tolerant language. As a general principle, errors are not reported unless absolutely necessary. This can lead to problems that are very hard to trace and debug. For example, spelling errors in variable names implicitly result in the creation of a new variable with the misspelled name. This is usually not the desired behavior, especially when compounded by case distinction in identifiers. While such behavior enables the successful execution of code lines that contain spelling errors, this usually results in other, significantly more difficult errors later in the execution. When an error is finally reported, the actual problem hides elsewhere in the program. Similarly, access to non-existent properties is allowed.

Other examples of the permissiveness of JavaScript include missing return statements: If the programmer forgets a return statement from a function, the execution of the program will continue with unexpected return values. Likewise, minor accidental syntax errors, such as using square brackets "`[]`" instead of parentheses "`()`" in the "`String.charAt()`" string indexing function will go unreported and can lead to problems that are very difficult to trace. Finally, almost anything in JavaScript, even system features, can be overridden. For example, the ability to override some of the string manipulation operations can be convenient, but can also lead to unanticipated behavior especially in those situations in which multiple programmers are working on the same application.

Lack of modularity. JavaScript 1.x does not include any syntactic notions for defining modules or classes explicitly. Rather, the programmer is expected to define (the equivalent of) classes in a prototype-based fashion [NTM99], using functions that are attached to an object's prototype. For modularity, JavaScript objects can be represented in a tree structure familiar from other prototype-based systems such as Self, NewtonScript [Smi95] or Kevo [Tai92]. However, in the absence of information hiding capabilities (see the next paragraph), such a structure provides only limited modularity. Furthermore, with the current crop of JavaScript VMs, there is a substantial performance penalty in referring to objects that are contained deep in the object tree. It should be noted that some variants of JavaScript such as ActionScript [Moo07] or proposed new versions of JavaScript, especially JavaScript 2.0, do provide mechanisms for defining classes explicitly. Furthermore, many JavaScript libraries such as the Dojo toolkit (www.dojotoolkit.org) or the Prototype library (www.prototypejs.org) enhance the Core JavaScript language with syntactic conventions for defining modules and classes.

No information hiding capabilities. Information hiding is a principle that goes hand in hand with modularity and well-defined interfaces. Basically, in order to isolate design decisions and implementation-level issues from the external use of a software component, the internals of the component should be hidden and preferably represented separately from the interface. Unfortunately, JavaScript 1.x does not provide any support for distinguishing between public or private functions and data. Such mechanisms have been proposed for JavaScript 2.0, but they are not available in the widely used versions of the JavaScript language.

Syntactic issues. Various issues related to JavaScript syntax were encountered during our project. First, JavaScript syntax is redundant. For instance, there are three different ways to define a function. Second, the need to use '`this`' for referring to all the instance variables can be annoying. Even though the excessive use of '`this`' can be avoided using the '`with`' statement, programs do not look as clean as the corresponding Java applications. Furthermore, as we already mentioned earlier, minor syntactic changes and accidental syntax errors can cause unexpected changes in the behavior of the program. For instance, creating a new object accidentally using the "`[]`" (array creation) operation instead of the "`{}`" (object creation) operation can result in unanticipated behavior later in the program. Because of the permissiveness and error-tolerance of the JavaScript language, such problems can go unnoticed for a long time.

On the positive side, JavaScript syntax is so similar to C, C++ and Java that we have found it very easy to port existing C, C++ and Java applications to JavaScript. Moreover, instantiation of objects with pre-initialized values in JavaScript (using the "`{ name: value, ... }`" syntax) is far more convenient than in C++ or in the Java programming language.

4.2 Library comments

Compared to languages such as Smalltalk or Java, JavaScript libraries are still relatively immature and incomplete. Furthermore, we have found the default "Client-Side" JavaScript I/O model unnecessarily complex and unsuitable for our purposes. Below we provide some comments and observations about JavaScript libraries.

Core JavaScript libraries are incomplete. Core JavaScript libraries are not as comprehensive as those of the Java programming language or many other languages. Several essential or even fundamental functions – such as object copying (an operation that is of critical importance in a prototype-based language!) – are missing from the JavaScript 1.5 libraries. In general, the functionality of the Core JavaScript libraries is significantly more limited than the functionality of core Java libraries.

No standardized libraries for advanced networking, graphics or media. Unlike with Java, there are no standardized JavaScript libraries for various important areas such as advanced networking, 2D/3D/vector graphics, audio, video or other advanced media capabilities. Even though such libraries have been defined as part of external JavaScript library development activities such as Dojo (www.dojotoolkit.org), no officially accepted standards for these areas exist yet. This is in contrast with the Java programming language that has a vast collection of standardized libraries available, defined through the Java Community Process (www.jcp.org).

The Client-Side JavaScript I/O model is overly complex and clumsy to use. On the web today, the communication between a scripting language such as JavaScript and the web browser revolves around the HTML markup language and the Document Object Model (DOM). DOM is a platform-independent way of representing a collection of objects that constitute a page in a web browser. The DOM allows a scripting language to programmatically examine and change the web page. The DOM serves as an interface – effectively a large shared data structure – between the browser and the scripting language, allowing the two to communicate with each other flexibly.

A set of "Client-Side" JavaScript APIs has been standardized to perform browser-based I/O from JavaScript. Unfortunately, these Client-Side JavaScript APIs are overly complex and clumsy to use. This is partly because communication between the scripting language and the browser takes place in the form of *side effects*: the scripting language tweaks the DOM tree, and – as a side effect – the browser will pick up the changes and update its display. In software engineering, the use of such side effects has been discouraged for decades. Alternatively, the scripting engine can construct HTML pages as strings on the fly and then send those strings to the browser with the expectation that the browser will update its screen accordingly. Such a model is also very awkward (and unnecessarily slow!) compared to traditional desktop systems in which a programming language can manipulate the screen directly by using a graphics library supporting direct drawing and direct manipulation.

4.3 Software configuration comments

The Core JavaScript language lacks certain mechanisms that would be valuable in creating and managing large applications that consist of multiple modules or subsystems. We refer to our comments in this area broadly as software configuration comments.

No 'include' or 'load' directive in Core JavaScript. Core JavaScript does not provide any mechanisms for loading in additional JavaScript code while an application is already running. Such capabilities would be useful, for instance, for system applications that need to load in additional system code or launch further applications from an environment written JavaScript. External JavaScript libraries often include support for this; for instance, the Dojo toolkit has a 'require' construct that allows an application to load in additional JavaScript code as necessary.

No support for unloading parts of a program. In addition to the 'include' or 'load' functionality, it would be equally valuable for a long-running application to be able to dispose of its parts or modules explicitly. Such unloading capabilities would be essential in a long-running system that relies on modules that can be upgraded (by unloading and then reloading the modules) without shutting down the system. In principle, such behavior can be emulated by using JavaScript's 'delete' operation or by carefully utilizing the garbage collector: By removing all the references to a certain branch in a program, the garbage collector should be able to remove all the objects under that branch. However, the effective use of such techniques requires the programmer to structure the programs in a certain fashion that may not always be feasible or optimal. Therefore, it would be better to have explicit mechanisms available for loading and unloading modules.

Loading multiple JavaScript applications into the same virtual machine is problematic. Because JavaScript does not provide support for defining modules or classes explicitly, loading multiple applications for execution in the same virtual machine can be problematic. For instance, if two programs happen to use the same names for global variables or functions, the overlapping variables or functions of the first program will be replaced with the corresponding features of the second program, resulting in unexpected behavior in the subsequent execution of the first program. Since JavaScript will not typically give any warning messages or errors in such situations, the resulting behavior can sometimes be a total surprise; in many cases the errors resulting from such situations may not be discovered until much later.

4.4 Development style comments

JavaScript is a highly dynamic language that requires a programming style that is quite different from the style used with more static languages such as C, C++ or the Java programming language. Below we provide some comments and observations about development style.

Evolutionary development approach is a necessity. Due to the highly permissive, error-tolerant nature of JavaScript, JavaScript programming requires an incremental, evolutionary software development approach. Since errors are reported much later than usual, by the time an error is reported it is often surprisingly difficult to pinpoint the original location of the error. Error detection is made harder by the dynamic nature of JavaScript, for instance, by the possibility to change some of the system features on the fly. Furthermore, in the absence of strong, static typing, it is quite possible to execute a program and only at runtime realize that some parts of the program are not yet present. For all these reasons, the best way to write JavaScript programs is to proceed step by step, by writing (and immediately testing) each new piece of code. If such an incremental, evolutionary approach is not used, debugging and testing can become quite tedious even for relatively small JavaScript applications. In general, the programming style required by JavaScript is closer to the exploratory programming style used in the context of other dynamic programming languages such as Smalltalk or Self.

Code coverage testing is important. The dynamic, interactive nature of JavaScript makes testing deceptively easy. In the presence of an interactive command shell and the 'eval' function, each piece of code can be tested immediately after it has been written. However, the use of such immediate testing approach does not guarantee the program to be bug-free or complete. In a static programming language, many simple errors will be caught already during the compilation of the program. However, in a dynamic language, it is not possible to know statically if a piece of code that has never been executed will actually run without problems. Since programs may contain numerous rarely executed branches – for instance, exception handlers – code coverage testing is very important. *Code coverage* measures to which degree the source code of a program has been tested. Code coverage is commonly measured as a percentage of the source code that has undergone execution and testing. Any dynamic program with less than 100% code coverage testing can still contain undiscovered problems. Even with 100% code coverage, it is still possible that further problems will be found.
Completeness of applications is difficult to determine. The flexibility of dynamic languages relies heavily on *late binding*: the references between various components of a program are resolved at runtime, as opposed to static languages in which such references are bound statically by the linker. As a result of their flexibility, dynamic languages make it easy to get applications up and running quickly. Compared with static programming languages that do not allow the execution of incomplete applications, applications written using a dynamic language can be run as soon as even some rudimentary code has been written. However, this flexibility has a price. With dynamic languages, it is easy to write incomplete applications that contain various undiscovered problems that will be found much later, only after the application has been executed several times. In general, with JavaScript as well as with other dynamic languages, it is nearly impossible to know if all the necessary pieces of an application are present before actually running the application. Static verification tools (such as 'jslint' for JavaScript) could play an important role in helping analyze the integrity of the application ahead of execution. To some extent, integrated development environments can also help in guiding the programmer through the most treacherous waters.

Event-oriented programming model works surprisingly well. The JavaScript language does not have built-in support for threads. Consequently, in standard JavaScript it is impossible to define applications that would have multiple threads of program control executing simultaneously. Instead, the programmer is expected to use an *event-oriented* programming model. In this model, each external occurrence such as a user interface activity (for instance, the user moving or clicking the button of a mouse) or network event (for instance, a completed HTTP request) is defined as an event that has an associated event handler *callback* function that will be invoked when the event triggers. At any point in time, only a single event handler callback function can be executing, that is, from the JavaScript programmer's viewpoint there is always only a single thread of control. There may be numerous native threads executing simultaneously, for instance, to process pending I/O requests, but they are not visible at the JavaScript level.

The event-oriented approach simplifies end-user programming considerably. For instance, there is much less reason to worry about race conditions or deadlocks, which are common concerns in those applications that are built around threads. In general, we find the event-oriented programming model more intuitive especially for novice programmers. The event-oriented model has some limitations, though. For instance, the event-oriented model is not ideal for systems programming if there is a need to run several applications simultaneously. Also, since each event handler callback function must complete its execution before another callback can be processed, the programmer has to design each callback function to be relatively short-lived in order to avoid delays in event processing.

4.5 Virtual machine comments

Like programs written for most dynamic languages, JavaScript applications run on top of a virtual machine. A *virtual machine* is a computing architecture implementation that has been written entirely in software, allowing an application written for that computing architecture to run independently of the underlying hardware. In this section we provide some comments on our experiences with JavaScript virtual machines so far.

JavaScript virtual machines are very reliable. As part of our project, we have used a number of different JavaScript virtual machines (VMs), including Mozilla's SpiderMonkey and Rhino (http://www.mozilla.org/js), and Apple's KJS. All these VMs have proven to be very reliable and almost impossible to crash.

Virtual machine porting interfaces are surprisingly good. JavaScript VMs are generally designed to be embedded inside web browsers or other systems in which scripting capabilities are needed. JavaScript VM designers have usually paid a lot of attention to the ease of embedding their VMs in new environments. We have found both SpiderMonkey and Rhino easy to use and port into different kinds of environments. The mapping of functions and events between the VMs and the host system has turned out to be simple.

A lot of room is left in optimizing JavaScript performance. Current JavaScript virtual machines are unnecessarily slow. Even though JavaScript is a significantly more dynamic language than, for instance, the Java programming language, there is no fundamental reason for JavaScript VMs to run two orders of magnitude slower than Java virtual machines. At the very minimum, JavaScript VM performance should be comparable to optimized Smalltalk VM implementations. This is not yet the case. Fortunately, a number of higher-performance JavaScript VMs are on their way, including Mozilla's *Tamarin* virtual machine (http://www.mozilla.org/projects/tamarin/).

Memory management capabilities of the current JavaScript VMs are poorly suited to large, long-running applications. Current JavaScript virtual machines have simple, 1970's style garbage collectors and memory management algorithms that are poorly suited to large applications. For instance, with large applications that allocate tens of megabytes of memory, garbage collection pauses on the SpiderMonkey VM can be excessively long; occasionally such pauses can take tens of seconds even on a fast machine. Furthermore, the SpiderMonkey VM makes dynamic memory allocation and deallocation (malloc and free) requests to the underlying operating system much too frequently. As in the VM performance area, with modern virtual machine implementation techniques memory management behavior could be improved substantially.

4.6 Other comments and observations

In this subsection we provide some additional comments and observations that do not belong to any of the categories discussed above.

JavaScript can be viewed as a multi-paradigm programming language. As we already noted earlier, we have found it easy to port existing C, C++ or Java applications into JavaScript. In fact, many C and Java applications can be translated to JavaScript almost mechanically. In our porting activities, we have observed that people end up using JavaScript in several different ways. Those applications that were translated from C often end up looking much like C programs. Likewise, a Java program translated into JavaScript tends to reflect the structure and the semantics of the original Java program. In contrast, when developing new JavaScript applications from scratch, it is possible to choose from a number of development styles, including a prototype-based object-oriented style familiar from Self [UnS87] or a functional development style. Based on our observations, JavaScript can be classified as a *multi-paradigm* programming language. By multi-paradigm, we mean that JavaScript can fairly easily accommodate a number of programming paradigms and allow programs to be written using different styles.

Applications tend to be shorter than expected. In our development activities, JavaScript has turned out to be a surprisingly expressive language. Compared to applications written in C, C++ or Java, JavaScript applications are often unexpectedly short. For instance, the core of the Morphic system consists of about 40 pages of code, including a sophisticated graphical user interface that includes a bracket-matching JavaScript code editor.

Limits of JavaScript as a real programming language were not reached. Dynamic languages have often been criticized for being unsuitable for the development of "serious" applications. Our development activities so far do not confirm this observation. Even though the debugging of large JavaScript applications can be painful, we have not had any significant trouble in managing JavaScript applications that consist of up to fifteen thousand lines of source code, the largest of our applications so far. In general, the applications that we have developed have not become so complex that they would have become unmanageable, even though we have had a number of different people working on the same source code. Further development work is needed to gather experience with even larger applications.

5. Conclusions

In this paper, we have described our experiences in using JavaScript as a real programming language in the context of the Lively Kernel – a web programming environment that supports highly dynamic, interactive web applications. As part of our project, we have written a lot of JavaScript code and applications that exercise the JavaScript language in a different fashion than the typical JavaScript programs found on commercial web sites. Among other things, we have used JavaScript as a systems programming language to write the Lively Kernel system itself.

In this brief report, we have scratched only the surface of our experiences and comments related to JavaScript. To summarize our observations, the extreme permissiveness and error tolerance of the language can make application development and debugging quite challenging. Core JavaScript libraries are not as mature as those of other languages such as Smalltalk or the Java programming language. Furthermore, JavaScript virtual machines, while generally very reliable, are not as efficient as virtual machines for many other dynamic languages. There are also some syntactic quirks in the JavaScript language that must be kept in mind when transitioning from syntactically similar programming languages such as C, C++ or Java. Nevertheless, we have found JavaScript to be a compelling programming language, both when porting existing C, C++ or Java applications and when creating entirely new applications. The expressiveness of the JavaScript language makes it well suited for building new applications by rapid prototyping.

In summary, we have found the JavaScript language to be a lot of fun, much more so than its reputation as a "toy" language might suggest. Based on our experiences, JavaScript is a surprisingly powerful and expressive language. Our experiences suggest that the JavaScript language can be used for developing real applications and even system software. However, effective use of JavaScript requires adjustment especially from those developers who are accustomed to more static programming languages. In general, JavaScript requires an incremental, evolutionary software development approach that is not yet familiar to the majority of software developers, except for those who have grown up with dynamic languages. As the evolution of the World Wide Web drives more and more software developers to use dynamic languages, the problems associated with such a development style will eventually disappear.

References

CPJ05 Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.

ECM99 ECMA Standard 262: ECMAScript Language Specification, 3rd edition, December 1999. Web link: http://www.ecma-international.org/publications/standards/Ecma-262.htm

Fla06 Flanagan, D., *JavaScript: The Definitive Guide*. O'Reilly Media, 2006.

GoR83 Goldberg, A., Robson, D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

Goo06 Goodman, D., *Dynamic HTML: The Definitive Reference*. O'Reilly Media, 2006.

IKM97 Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. Presented at the OOPSLA'97 Conference. Web link: http://ftp.squeak.org/docs/OOPSLA.Squeak.html

Mal95 Maloney, J.H., Morphic: The Self User Interface Framework. Self 4.0 Release Documentation, Sun Microsystems Laboratories, 1995.

MaS95 Maloney, J.H., Smith, R.B., Directness and liveness in the morphic user interface construction environment. Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, 1995, pp. 21-28.

Moo07 Moock, C., Essential ActionScript 3.0. O'Reilly Media, 2007.

NTM99 Noble, J., Taivalsaari, A., Moore, I. (eds), Prototype-Based Programming: Concepts, Languages and Applications. Springer Publishing Company, Singapore, 1999.

Pra07 Prabhakar, C., *Google Web Toolkit: GWT Java Ajax Programming*. Packt Publishing, 2007.

Pau07 Paulson, L.D., Developers shift to dynamic programming languages, *IEEE Computer*, vol 40, nr 2, February 2007, pp. 12-15.

Smi95 Smith, W.R., Using a Prototype-based Language for User Interface: The Newton Project's Experience. In OOPSLA'95 Conference Proceedings (Austin, Texas, October 15-19), ACM SIGPLAN Notices vol 30, nr 10 (Oct) 1995, pp. 61-72.

Tai92 Taivalsaari, A., Kevo – a prototype-based object-oriented language based on concatenation and module operations. University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992.

Tat06 Tate, B., *Ruby on Rails: Up and Running*. O'Reilly Media, 2006.

UnS87 Ungar, D., Smith, R.B., Self: the power of simplicity. In OOPSLA'87 Conference Proceedings (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987, pp. 227-241.

About the Authors

Dr. Tommi Mikkonen is a Visiting Professor at Sun Labs, and a professor of computer science at Tampere University of Technology, Finland. Tommi is a well-known expert in the area of software engineering. He has arranged numerous courses on software engineering and mobile computing, and he just recently published a book on mobile software development with Wiley & sons.

Dr. Antero Taivalsaari is a Principal Investigator and Senior Staff Engineer at Sun Labs. Antero is best known for his work on the Java™ Platform, Micro Edition (Java ME). Before returning back to Sun Labs in August 2006, Antero spent seven years working in Sun's Java Software organization, witnessing the evolution of the Java ME platform from a two-person research project to one of the most popular commercial software platforms in the world, with over a billion Java ME devices worldwide. Antero has received Sun's Chairman's Award twice (in 2000 and 2003) for his work on Java ME technology.