Journal of Universal Computer Science, vol. 4, no. 5 (1998), 522-533
submitted: 14/11/97, accepted: 8/1/98, appeared: 28/5/98 © Springer Pub. Co.

Open Standards Beyond Java: On the Future of Mobile Code for the Internet[1]

Michael Franz
(Department of Information and Computer Science,
University of California at Irvine, USA
franz@uci.edu)

Abstract: At first sight, Java's position as the de-facto standard for portable software distributed across the Internet seems virtually unassailable. Interestingly enough, however, it is surprisingly simple to provide alternatives to the Java platform, using the plug-in mechanism supported by the major commercial World Wide Web browsers.

We are currently developing a comprehensive infrastructure for mobile software components. This is a long-term research activity whose primary objectives are not directly related to today's World Wide Web, but which targets future high-performance component-software systems. However, purely as a technology demonstration, we have recently started a small spin-off project called "Juice" with the intent of extending our experimental mobile-code platform into the realm of the commercial Internet.

Juice is implemented in the form of a browser plug-in that generates native code on-the-fly. Although our software distribution format and run-time architecture are fundamentally different from Java's, and arguably more advanced, once that the appropriate Juice plug-in has been installed on a Windows PC or a Macintosh computer, end-users can no longer distinguish between applets that are based on Java and those that are based on Juice. The two kinds of applets can even coexist on the same Web-page.

This, however, means that Java can in principle be complemented by alternative technologies (or even gradually be displaced by something better) with far fewer complications than most people seem to assume. As dynamic code generation technology matures further, it will become less important which code-distribution format has the largest "market share"; many such formats can be supported concurrently. Future executable-content developers may well be able to choose from a wide range of platforms, probably including several dialects of Java itself. Hence, a pattern of "open standards" for mobile code is likely to eventually emerge, in spite of Java's current dominance.

Key Words: Open standards, mobile code technologies, alternatives to Java, plug-in browser extensions, on-the-fly code generation, Java, Juice, Oberon.

1 Introduction

One of the most beneficial aspects of the rapid expansion of the Internet is that it is driving the deployment of "open" software standards. We are currently witnessing the introduction of a first suite of interoperability standards that is already having far-reaching influences on software architecture, as it simultaneously also marks the

[1] This is an extended version of a paper presented at WebNet'97. The paper was judged as outstanding and has received a "Best Paper Award".

transition to a *component model* of software. The new standards, such as *CORBA* (Object Management Group), *COM/OLE* (Microsoft), and *SOM/OpenDoc* (Apple Computer, IBM, Novell), enable software components to inter-operate seamlessly, even when they run on different hardware platforms and have been implemented by different manufacturers. Over time, the monolithic application programs of the past will be supplanted by societies of inter-operating, but autonomous, components.

It is only logical that the next development step will lead to even further "openness", not only freeing components from all dependence upon particular hardware architectures, but also giving them the autonomy to migrate among machines. Instead of executing complex transactions with a distant server by "remote control" over slow communication links, software systems will then be able to send to a server self-contained mobile agents that complete the transactions autonomously on the user's behalf. The inclusion of *executable content* into electronic documents on the World Wide Web already gives us a preview of how powerful the concept of mobile code is, despite the fact that so far only a unidirectional flow of mobile programs from server to client is supported. Distributed systems that are based on freely-moving agents will be even more powerful.

In order to transfer a mobile program between computers based on different processor architectures, some translation of its representation has to occur at some point, unless the mobile program exists in multiple execution formats simultaneously. Although the latter approach seems feasible in the current context of software distribution via CD-ROM, its limits will soon become apparent when low-bandwidth wireless connectivity becomes pervasive. Hence, a *universal* representation for mobile code is required. The search for such a universal representation is the subject of much current research [Engler 1996, Inferno, Lindholm and Yellin 1996], including recent work of the author [Franz & Kistler 1997, Kistler & Franz 1997].

In the short time since its launch, Sun Microsystems's *Java* technology has become almost synonymous with portable software that can be distributed across the Internet. Java's pre-eminent position is reinforced by the fact that built-in support for its distribution format, the *Java Virtual Machine* (JVM), is now not only part of practically every World Wide Web browser, but is starting to appear even within operating systems. Yet in spite of the de-facto adoption of Java by most of the Internet community as the standard platform for encoding executable content (at least for the time being), it remains surprisingly simple to provide alternatives to this platform, even within the context of commercial browser software.

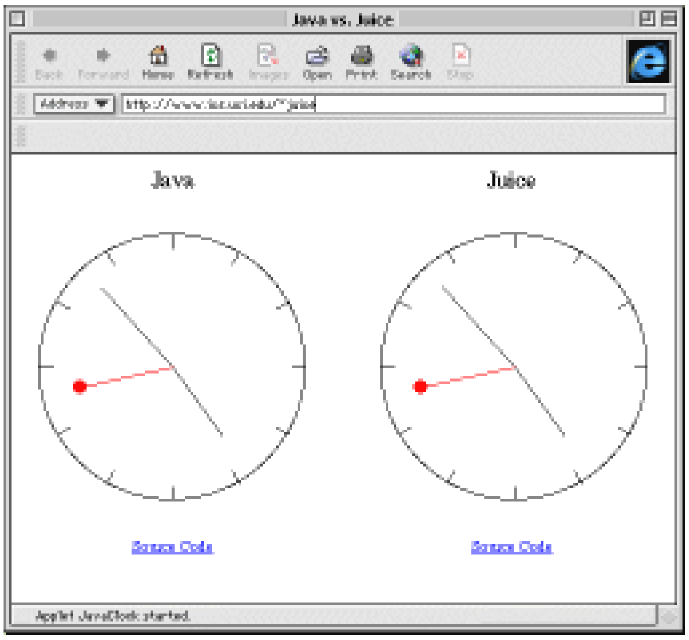
We have created such an alternative to the Java platform and named it "Juice". Juice is an extension of the first author's earlier research on portable code and on-the-fly code generation [Franz & Ludwig 1991, Franz 1994a, Franz 1994b][2]. Our current work is significant on two accounts: First, Juice's portability scheme is technologically more advanced than Java's and may lead the way to future mobile-code architectures. Second, the mere existence of Juice demonstrates that Java can be complemented by alternative technologies with far less effort than most people seem to assume. In fact, once that Juice has been installed on a machine, end-users need not be concerned at all

[2] Note that this earlier work on mobile code predates Java by several years

whether the portable software they are using is based on Juice or on Java. In light of this, we are surprised by the widespread belief in the myth that, in order to be portable, all executable content must necessarily be encoded in Java. In the short term, this myth may lead to some ill-founded technology decisions.

Just as with Java, there are three major components to the Juice technology: 1) a source language and an API in which Juice applets are programmed, 2) an architecture-neutral distribution format, and 3) an environment for executing Juice applets, which in the current implementation of Juice is supplied in the form of a browser plug-in that *generates native code on-the-fly*. On all three accounts, Juice differs considerably from Java, yet from the Web-browsing end-user's perspective, there is no obvious difference between Java and Juice applets.

In the following sections, we will introduce the three components of the Juice platform: source language, distribution format, and dynamic-compilation environment, which not only provides on-the-fly code generation, but also dynamic code re-optimization in the background. We will conclude by presenting examples of Juice and Java source-code side-by-side, arguing that the choice of a particular mobile-code solution may simply be a matter of personal taste, rather than a technological necessity. Luckily, it is the applet developer that needs to make this choice; the end user need not know any of it as multiple mobile-code technologies, such as Java and Juice, can happily coexist, even on the same Web page [Fig. 1].



Applet JavaClock started.

Figure 1: Java Applet (left) vs. Juice Applet (right)

2 The Source Language of the Juice Platform is Oberon

Juice applets are programmed in the language *Oberon* [Wirth 1988], a direct successor of Pascal and Modula-2 that was defined by Niklaus Wirth (Pascal's original creator) in 1988. Oberon is surprisingly close to Java in spirit; like Java, Oberon is based on the principles of simplicity and safety. Oberon enforces type-safety by mandating array-bounds checking and prohibiting pointer arithmetic, it automates memory management through the provision of garbage collection, and provides source-level modularization facilities along with dynamic loading. Superficially, but not entirely untrue, one might argue that Oberon is a subset of Java with Pascal syntax, except that Oberon was defined several years before Java.

Oberon is a much smaller language than Java, having been designed almost as the "essence of a programming language". For example, Oberon provides no language-level support for concurrency. While we agree with Ted Lewis [Lewis 1997] that Java's concurrency scheme falls disappointingly short of the existing state-of-the-art ante, Oberon offers no built-in support for concurrency at all. For the project described here, we have not attempted to change the Oberon language and have therefore only studied applets that can be constructed from the set of language features in the intersection of Oberon and Java. However, we note that this is only an incidental effect of our choice of Oberon as a source language and in no way limits our claim about Java's substitutability in principle. Moreover, we note that some current optimizing translators for Java also exclude the concurrency capabilities of the Java language [Muller et al. 1997], because support of threads has a performance penalty associated with it [Proebsting et al. 1997].

A Juice-applet development tool-kit is now a standard part of the Oberon software distribution [Oberon] from ETH Zurich and UC Irvine for Apple Macintosh and Microsoft Windows. Besides providing a full implementation of *Oberon System 3* [Wirth & Gutknecht 1989, Wirth & Gutknecht 1992, Gutknecht 1994], it supplies a set of Juice-specific APIs along with a compatibility-box recreating the environment of a browser plug-in within the Oberon environment. Hence, Juice applets under construction can be tested interactively without having to exit the development environment.

3 Juice Applets are Distributed as Slim Binaries

Juice's mobile-code architecture is based on a software distribution format called *slim binaries* [Franz & Kistler 1997] that constitutes a radical departure from traditional software-portability solutions. Unlike the usual approach of representing mobile programs as instruction sequences for a virtual machine, an approach taken both with p-code [Nori et al. 1976] as well as with Java byte-code [Lindholm and Yellin 1996], the slim binary format is instead based on *adaptive compression of syntax trees* [Franz 1994a]. When compiling a source program into a slim binary, it is first translated into a tree-shaped intermediate data structure in memory that abstractly describes the semantic actions of the program (e.g., "add result of left sub-tree to result of right sub-tree"). This data structure is then compressed by identifying and merging isomorphic sub-trees, turning the tree into a directed acyclic graph with shared sub-trees (for example, all occurrences of "x + y" in the program could be mapped onto a single sub-tree that represents the sum of "x" and "y"). The linearized form of this graph constitutes the slim binary format.

In the actual implementation, tree compression and linearization are performed concurrently, using a variant of the classic LZW data-compression algorithm [Welch 1984]. Unlike the general-purpose compression technique described by Welch, however, our algorithm is able to exploit domain knowledge about the internal structure of the syntax tree being compressed. Consequently, it is able to achieve much higher information densities [Fig. 2]. We know of no conventional data-compression algorithm, regardless of whether applied to source code or to object code (for any architecture, including the Java virtual machine), that can yield a program representation as dense as the slim binary format.

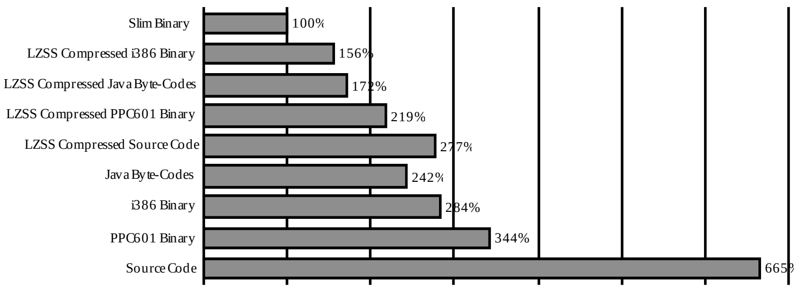


Figure 2: Relative Size of a Representative Program Suite in Various Formats

The compactness of the slim binary format may soon become a major advantage, as many network connections in the near future will be wireless and consequently be restricted to small bandwidths. In such wireless networks, raw throughput rather than network latency again becomes the main bottleneck. We also note that one could abandon native object code altogether in favor of a machine-independent code format if the portable code would not only *run* as fast as native code, but also *start up* just as quickly (implying that there would be no discernible delay for native-code translation). As the author has shown in previous work, this becomes possible if the portable software distribution format is so dense that the additional computational effort required for just-in-time code generation can be compensated entirely by reduced I/O overhead due to much smaller "object files" [Franz 1994a, Franz 1994b, Franz 1997a].

Compactness does come at a small price: since isomorphic sub-trees have been merged during encoding, a program represented in the slim binary format cannot simply be interpreted byte-by-byte. While representations such as p-Code and Java byte-codes permit random access, i.e. one can jump 20 instructions forward in the code and resume interpretation, this is not possible with slim binaries. Each symbol in a slim-binary-encoded program can be interpreted only in the context of all the symbols that precede it. Because of this characteristic, our implementations have eschewed interpretation of the intermediate form from the very beginning and have incorporated on-the-fly code generators.

On the other hand, reading a slim binary in our system re-creates the original tree data-structure, which is an almost ideal input for an optimizing code-generator. The slim binary format preserves structural information such as control flow and variable scope that is lost in the transition to linear representations such as Java byte-codes. In order to perform code generation with advanced optimizations from a byte-code representation, a time-consuming pre-processing step is needed to rediscover the lost structural information. This is not necessary with slim binaries. This argument applies not only with respect to code optimization, but also for *code verification*: analyzing a mobile program for violation of type and scoping rules is much simpler when the program has a tree-based representation than it is with a linear byte-code sequence.

Our present implementation of slim binaries is in so far restrictive as it supports exactly one source language, Oberon. In this respect, our system presently doesn't do much better than abstract-machine-based portability schemes in which the instruction set of the virtual machine is explicitly crafted to support a particular source language. While we do not foresee any difficulties in encoding syntax trees for other languages, possibly even using the identical format, the suitability for other languages has yet to be established by an actual implementation. We have therefore recently started a follow-up project with the aim of constructing a compiler that takes Java as its input, but generates slim binaries instead of Java byte-codes as its output. This tool will not only enable a more direct comparison of our code representation and our dynamic compilation architecture with their respective Java counterparts, but it will also aid the wider discussion of platform-independent mobile-code solutions by disengaging the question of source languages from the separate issue of finding suitable distribution formats.

4 Juice Applets are Compiled On-The-Fly by a Browser Plug-In

The only part of Juice that is visible to end-users is a set of platform-specific plug-ins for *Netscape Navigator* and *Microsoft Internet Explorer*. Once the appropriate Juice plug-in has been installed on a user's machine, the user can then view and execute Juice content in the same manner as Java applets. Hence, after installation of the plug-in, users can no longer distinguish between Java and Juice applets, other than by disabling either format manually.

The Juice plug-in contains a dynamic code-generator that translates from the slim binary representation into the native code of the respective target machine (PowerPC or Intel 80x86). This translation occurs before the applet is started, but is fast enough not to be noticed under normal circumstances. In contrast, most just-in-time compilers for Java translate individual methods as they are called rather than the whole applet at once. To Juice's advantage, translating the whole applet at once usually results in better code quality since it permits inter-procedural optimizations to be exploited.

Due to the greater compactness of slim binaries in comparison to Java byte-codes, less time has to be spent on the transmission of Juice applets. The time saved can then be used to offset the cost of code generation. As the speed of processors is rising faster than the speed of I/O, hardware technology is actually evolving in favor of denser code-representation formats, even if more work is required to “unpack” their contents.

The main thrust of our continuing research is focused on improving code quality. Our implementations so far are all based on a well-established family of compiler back-ends originating at ETH Zurich that produce high quality code comparable to that of straightforward commercial compilers [Brandis et al. 1995]. On some newer RISC architectures, however, these back-ends cannot fully compete with highly optimizing compilers. Of further concern to our particular application of load-time code generation is the fact that optimizers for certain RISC architectures may have vastly different run-time characteristics than the compilers we have been using so far.

Consequently, we are now pursuing a two-tier strategy of code generation. Rather than compiling every module exactly once when it is loaded and then leaving it alone, we use a background process executing only during idle cycles that keeps compiling the already loaded modules over and over [Kistler 1997, Franz 1997b]. Since this is strictly a re-compilation of already functioning modules, and since it occurs completely in the background, this process can be as slow as it needs to be, allowing the use of far more aggressive, albeit slower, optimization techniques than would be tolerable in an “interactive” context. When background code-generation has completed, the code-images of the re-generated modules are substituted for their older counterparts in situ, without disrupting the ongoing program execution.

Periodic re-optimization of already executing code allows to fine-tune the code-generator's output beyond the level commonly achievable by static compilation. Not only does it enable run-time profiling data from the current execution to drive the next iteration of code optimization, but it also makes it possible to cross-optimize application programs and their dynamically loaded extensions and libraries. We are currently experimenting with global optimization techniques that were pioneered by incremental compilers and link-time optimizers. Among them are register allocation and code inlining across module boundaries, global instruction scheduling, and global cache optimization. Run-time extensible systems present new challenges to these old problems, since no closed analysis is possible due to the fact that further modules can be dynamically linked to the already executing system at any time.

5 A Direct Comparison of Juice and Java Programming

The easiest way of demonstrating the different “flavors” of programming in Juice vs. programming in Java is by presenting actual source texts. In [Fig. 3] we present, side by side, the source of a simple applet displaying the current time in analog form (as shown in [Fig. 1]), encoded using Java (left) and Juice (right). These sources and the resulting executable applets can also be found on our World Wide Web site.

```java
import java.awt.*;
import java.util.*;

public class JavaClock
extends java.applet.Applet
implements Runnable

{
    Thread timer = null;

    public void run()
    {
        while (timer!=null) {
            repaint();
            try {Thread.sleep(1000);}
            catch(InterruptedException e) {return;}
        }
    }

    public void start()
    {
        if (timer == null) {
            timer = new Thread(this);
            timer.start();
        }
    }

    public void stop()
    {
        timer = null;
    }

    public int min(int a, int b)
    {
        if (a < b) return a;
        else return b;
    }

    public void arcline(Graphics g, int angle, int x, int y, int r1, int r2, boolean dot)
    {
        int x1, y1, x2, y2; double s, c, a;

        angle = (angle - 15) % 60;
        a = 2*Math.PI / 60 * angle;
        s = Math.sin(a); c = Math.cos(a);
        x1 = (int)(r1*c + 0.5);
        y1 = (int)(r1*s + 0.5);
        x2 = (int)(r2*c + 0.5);
        y2 = (int)(r2*s + 0.5);
        g.drawLine(x+x1, y+y1, x+x2, y+y2);
        if (dot) g.fillOval(x+x2-5, y+y2-5, 10, 10);
    }

    public void paint(Graphics g)
    {
        int r, r0, rs, rm, rh, x, y, i;

        r = min(size().width, size().height) / 2; r0 = 10*r / 11;
        rs = 8*r /11; rm = 9*r/11; rh = 7*r/11; x = r; y = r;
        g.setColor(Color.white);
        g.fillRect(0, 0, size().width, size().height);
        g.setColor(Color.black);
        g.drawOval(0, 0, 2*r, 2*r);
        for (i=0; i<60; i+=5) arcline(g, i, x, y, r0, r, false);

        Date now = new Date();
        arcline(g, now.getMinutes(), x, y, 0, rm, false);
        arcline(g, now.getHours()*5+now.getMinutes()/12, x, y, 0, rh, false);
        g.setColor(Color.red);
        arcline(g, now.getSeconds(), x, y, 0, rs, true);
    }

    public void update(Graphics g)
    {
        paint(g);
    }

}
```

```
MODULE JuiceClock;

    IMPORT
        Math := JuiceMath, Applets := JuiceApplets,
        Devices := JuiceDevices, Misc := JuiceMisc;

    TYPE
        Applet = POINTER TO AppletDesc;
        AppletDesc = RECORD (Applets.AppletDesc)
            hour, min, sec: INTEGER
        END;

    PROCEDURE Min(a, b: INTEGER): INTEGER;
    BEGIN
        IF a < b THEN RETURN a ELSE RETURN b END
    END Min;

    PROCEDURE ArcLine(angle, x, y, r1, r2: INTEGER; dot: BOOLEAN);
        VAR x1, y1, x2, y2: INTEGER; s,c,a : REAL;
    BEGIN angle := (angle-15) MOD 60;
        a := 2 * Math.pi / 60 * angle;
        s := Math.Sin(a); c := Math.Cos(a);
        x1 := SHORT(ENTIER(r1*c + 0.5));
        y1 := SHORT(ENTIER(r1*s + 0.5));
        x2 := SHORT(ENTIER(r2*c + 0.5));
        y2 := SHORT(ENTIER(r2*s + 0.5));
        Devices.Line(x+x1, y+y1, x+x2, y+y2);
        IF dot THEN Devices.FillOval(x+x2-5, y+y2-5, 10, 10) END
    END ArcLine;

    PROCEDURE Update (me: Applet);
        VAR r, r0, rs, rm, rh, x, y, i: INTEGER;
    BEGIN Devices.Setup(me.device);
        r := Min(me.device.w, me.device.h) DIV 2; r0 := 10*r DIV 11;
        rs := 8*r DIV 11; rm := 9*r DIV 11; rh := 7*r DIV 11; x := r; y := r;
        Devices.SetForeColor(Devices.white);
        Devices.FillRect(0, 0, me.device.w, me.device.h);
        Devices.SetForeColor(Devices.black);
        Devices.FrameOval(0, 0, 2*r, 2*r);
        i := 0; WHILE i < 60 DO ArcLine(i, x, y, r0, r, FALSE); INC(i, 5) END;

        Misc.GetTime(me.hour, me.min, me.sec);
        ArcLine(me.min, x, y, 0, rm, FALSE);
        ArcLine(me.hour * 5 + me.min DIV 12, x, y, 0, rh, FALSE);
        Devices.SetForeColor(Devices.red);
        ArcLine(me.sec, x, y, 0, rs, TRUE);
        Devices.Restore(me.device)
    END Update;

    PROCEDURE AppletHandler (me: Applets.Applet; VAR M: Applets.AppletMsg);
        VAR hour, min, sec: INTEGER;
    BEGIN
        WITH me: Applet DO
            WITH M: Applets.DisplayMsg DO
                IF M.id = Applets.update THEN Update(me)
                ELSE Applets.AppletHandler(me, M)
                END
            | M: Applets.IdleMsg DO Misc.GetTime(hour, min, sec);
                IF (hour # me.hour) OR (min # me.min) OR (sec # me.sec) THEN
                    Update(me)
                END
            ELSE Applets.AppletHandler(me, M)
            END
        END
    END AppletHandler;

    PROCEDURE NewApplet*;
        VAR a: Applet;
    BEGIN NEW(a); a.handle := AppletHandler; Applets.newApplet := a
    END NewApplet;

END JuiceClock.
```

Figure 3: Java Source Code (left) vs. Juice Source Code (right)

6 Security Issues

From the very beginning, the question of security has played a large role in the discussion surrounding executable content for the World Wide Web. In particular, a number of security flaws were discovered in early implementations of Java that received a lot of publicity [Dean et al. 1996]. Needless to say that these errors were corrected as soon as they were discovered.

In principle, the topic of mobile-code security is independent of the choice of programming language that a mobile program is written in, and also independent of the representation that is used in transporting applets to target machines. A minimum requirement is only that *type-safety* is maintained. Both Java and Oberon are fully type-safe languages, and hence both are equally suitable for applet-programming.

However, absolute type-safety requires more than just compile-time checking because an attacker could hand-craft a malicious applet directly in the mobile code representation without passing it through a compiler, thereby circumventing the type-safety of the source language. As a consequence, mobile code needs to be scanned and *verified* prior to execution even if it is based on a “safe” language [Yellin 1995].

Most implementations of Java today provide such a verifier for incoming mobile code, while Juice currently does not. This is simply a restriction of the current implementation of the Juice system, and does not mean that Juice’s intermediate representation is less well suited for verification than Java’s. In fact, it is highly likely that Juice’s slim binary format will turn out to provide faster verification than Java’s virtual machine instructions.

This is because the verifier needs to perform an exhaustive data-flow analysis of the executable code. For each machine instruction, it must examine all possible paths leading to the instruction to ensure that registers hold values of the appropriate type. This analysis requires structural information about the program that is not immediately available in a sequential virtual-machine instruction stream. Extracting the required structural information from such a “flat” representation is a time-consuming task whose complexity grows faster than linear program size.

The slim binary representation, on the other hand, is tree-shaped and hence would not require this additional structural analysis phase. Since the time available for just-in-time code generation is usually limited by the patience of an interactive user, eliminating this time-consuming analysis is likely to result in faster-running native code because more effort can be devoted to other phases of code generation, such as instruction scheduling.

7 Conclusion and Outlook

Portable, executable content need not necessarily be tied to Java technology. In this paper, we have presented various aspects of a mobile-code infrastructure that differs from Java on several key accounts. Not only is our implementation a test-bed for novel code-representation and dynamic-compilation techniques, but it also confirms the suitability of the existing browser plug-in mechanism for supporting alternative software portability solutions.

As our "Juice" system demonstrates, the plug-in mechanism can even be utilized to provide on-the-fly native-code generation, enabling alternative portability schemes to compete head-on with Java in terms of execution speed. Our implementation also shows that alternative mobile code solutions can remain completely transparent to end-users once that an appropriate plug-in has been installed. Hence, the eventual migration path from Java to a successor standard at the end of Java's life-cycle will probably be much less painful than most people anticipate now.

In fact, the plug-in mechanism opens the door for potentially many different Java alternatives that could be introduced over time, *gradually* reducing Java's pre-eminence. Besides the Juice solution described here, a strong initial candidate to win market share from Java might be Lucent's *Inferno* [Inferno] (assuming that Inferno could be packaged as a plug-in), but other contenders will surely appear. Note that each plug-in itself can be distributed across the Internet, authenticated by a code-signing mechanism, simplifying the logistics of supporting several competing code-formats concurrently.

It is also possible, and even probable, that the Java standard itself will fragment into several dialects. For example, Microsoft is incorporating an API into its version of Java and its *Internet Explorer* browser that differs from the developments at Sun Microsystems, Java's original creator. There may come a point at which the differences between the various sets of libraries become irreconcilable, leading to mutually incompatible versions of Java. This difference could be hidden from end-users using the same approach that we have taken with Juice.

We believe that dynamic code generation technology is reaching a level of maturity that it will soon diminish the relative importance of "market share" of any particular code distribution format. In order to be commercially successful, distribution formats will have to mimic Java in providing architecture neutrality and safety, but further considerations such as code density will surely gain in importance. For example, some future distribution formats may be targeted towards particular application domains. On the other hand, Microsoft's ActiveX technology, which is based on a particular machine architecture, may still yet become a "universal" software standard, since dynamic code translation will soon be available to enable execution on incompatible hardware platforms. In this larger context, the current enthusiasm surrounding Java may soon appear to have been somewhat overblown.

Acknowledgement

The author gratefully acknowledges Thomas Kistler as the co-author of the Juice system. Part of this work is being funded by the National Science Foundation under grant CCR-9701400.

References

[Brandis et al. 1995] M. Brandis, R. Crelier, M. Franz, and J. Templ (1995); "The Oberon System Family"; *Software-Practice and Experience*, 25:12, 1331-1366.

[Dean et al. 1996] D. Dean, E. W. Felten, and D. S. Wallach (1996); "Java Security: From HotJava to Netscape and Beyond"; *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, California, 190-200.

[Engler 1996] D. R. Engler (1996); “Vcode: A Retargetable, Extensible, Very Fast Dynamic Code Generation System”; *Proceedings of the ACM Sigplan ‘96 Conference on Programming Language Design and Implementation*, published as *ACM Sigplan Notices*, 31:5, 160-170.

[Franz 1994a] M. Franz (1994); *Code-Generation On-the-Fly: A Key to Portable Software*; Doctoral Dissertation No. 10497, ETH Zürich, published in book form by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0.

[Franz 1994b] M. Franz (1994); “Technological Steps toward a Software Component Industry”; in J. Gutknecht (Ed.), *Programming Languages and System Architectures,* Springer Lecture Notes in Computer Science, No. 782, 259-281.

[Franz 1997a] M. Franz (1997); “Dynamic Linking of Software Components”; *IEEE Computer*, 30:3, 74-81.

[Franz 1997b] M. Franz (1997); “Run-Time Code Generation as a Central System Service”; in *The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI),* IEEE Computer Society Press, ISBN 0-8186-7834-8, 112-117.

[Franz & Kistler 1997] M. Franz and T. Kistler (1997); “Slim Binaries”; *Communications of the ACM*, 40:12, 87-94.

[Franz & Ludwig 1991] M. Franz and S. Ludwig (1991); “Portability Redefined”; in *Proceedings of the Second International Modula-2 Conference*, Loughborough, England.

[Gutknecht 1994] J. Gutknecht (1994), “Oberon System 3: Vision of a Future Software Technology”; *Software–Concepts and Tools*, 15:1, 26-33.

[Inferno] Lucent Technologies Inc.; *Inferno*; http://plan9.bell-labs.com/inferno/.

[Juice] M. Franz and T. Kistler; *Juice*; http://www.ics.uci.edu/~juice.

[Kistler 1997] T. Kistler (1997); “Dynamic Runtime Optimization”; in H. Mössenböck (Ed.), *Modular Programming Languages: Proceedings of the Joint Modular Languages Conference (JMLC’97)*, Springer Lecture Notes in Computer Science No. 1204, 53-66.

[Kistler & Franz 1997] T. Kistler and M. Franz (1997); “A Tree-Based Alternative to Java Byte-Codes”; *Proceedings of the International Workshop on Security and Efficiency Aspects of Java*, Eilat, Israel.

[Lewis 1997] T. Lewis (1997); “If Java is the Answer, What Was the Question?”; *IEEE Computer*, 30:3, 136&133-135.

[Lindholm and Yellin 1996] T. Lindholm and F. Yellin (1996); *The Java Virtual Machine Specification*; Addison-Wesley.

[Muller et al. 1997] G. Muller, B. Moura, F. Bellard, and Ch. Consel (1997); “Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code”; *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS),* USENIX Association Press, 1-20.

[Nori et al. 1976] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and Ch. Jacobi (1976); “Pascal-P Implementation Notes”; in D.W. Barron (Ed.); *Pascal: The Language and its Implementation*; Wiley, Chichester.

[Oberon] Department of Information and Computer Science, University of California at Irvine; *Oberon Software Distribution*; http://www.ics.uci.edu/~oberon.

[Proebsting et al. 1997] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson (1997); “Toba: Java For Applications – A Way Ahead of Time (WAT) Compiler”; *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS)*, USENIX Association Press, 41-53.

[Welch 1984] T. A. Welch (1984); “A Technique for High-Performance Data Compression”; *IEEE Computer*, 17:6, 8-19.

[Wirth & Gutknecht 1989] N. Wirth and J. Gutknecht (1989); “The Oberon System”; *Software-Practice and Experience*, 19:9, 857-893.

[Wirth & Gutknecht 1992] N. Wirth and J. Gutknecht (1992); *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley.

[Wirth 1988] N. Wirth (1988); “The Programming Language Oberon”; *Software-Practice and Experience*, 18:7, 671-690.

[Yellin 1995] F. Yellin (1995); “Low Level Security in Java”; Fourth International World Wide Web Conference, Boston, Massachusetts; World Wide Web Consortium; http://www.w3.org/pub/Conferences/WWW4/Papers/197.