The Newspeak Programming Platform

Gilad Bracha Peter Ahe Vassili Bykov Yaron Kashai
Eliot Miranda

Cadence Design Systems

May 6, 2008

Abstract

Cadence is developing a high productivity development platform based around the Newspeak programming language. Newspeak supports advanced capabilities not found in mainstream languages, such as mixins, virtual classes and class hierarchy inheritance.

Newspeak is based on an object capability model which provides a solid foundation for security. The language has a powerful module system designed to facilitate both development and deployment. The Newspeak platform includes an IDE, a GUI library, and standard libraries.

1 Introduction

The Newspeak platform consists of the Newspeak programming language and allied tools and libraries. This paper gives an overview of some of the platform's key features and reviews its current status. The design of the Newspeak platform is based on our experience maintaining and extending Java [13] and on our work with Smalltalk [7, 1].

The most important property of the Newspeak language is that it is a *message based* programming language. This means that all computation — even an object's own access to its internal structure — is performed by sending messages to objects. This implies that everything in Newspeak is an object, from elementary data such as numbers, booleans and characters up to functions, classes and modules.

Below is a simple example of a Newspeak class which introduces the class Point. The class defines two *slots* x and y. Slots are similar to instance variables, except that they are never accessed directly. The slots are accessed only through automatically generated getters and setters. If p is point, p x and p y denote the values stored in p's x and y slots respectively. Note that there is no dot between p and x; since method invocation is the only operation in Newspeak, it can be recognized implicitly by the compiler.

Setter methods are denoted by the slot name followed by a :, so p x: 91 sets the x coordinate of p to 91.

```
class Point = (
(* ClassCategory: Graphics kernel *)
(* A class representing points in 2-space *)
  | x y |  (* Declare slots *)
)
(
(* MessageCategory: printing *)
public printString = (
  ' x = ', x printString, ' y = ', y printString
))
```

Within the body of Point, the names x, y, x: and y:, can be used directly, as shown in the method printString. However, x and y denote calls to the getter methods, not references to variables. The comma method of String implements string concatenation.

Newspeak programs enjoy the property of *representation independence* - one can change the layout of objects without any need to make further source changes anywhere in the program. For example, if we choose to modify Point so that it uses polar coordinates, no modification to the printString method is needed, as long as we preserve the interface of Point by providing methods x and y that compute the cartesian coordinates (note that the caret (^) is used to indicate that an expression should be returned from the method, just like the **return** keyword in conventional languages):

```
class Point = (
(* ClassCategory: Graphics kernel *)
(* A class representing points in 2-space *)
  | rho theta |  (* Declare slots *)
)
(
(* MessageCategory: access *)
public x = ( ^rho * theta cos)
public y = (^rho * theta sin)
(* MessageCategory: printing *)
public printString = (
  ' x = ', x printString, ' y = ', y printString
))
```

Newspeak classes can be nested within one another to arbitrary depth. Thus, a Newspeak class can have three kinds of members: slots, methods and classes. All references to names are always treated as method calls, so any member declaration within a class can be overridden in a subclass. It is possible not only to override methods with methods (as with virtual methods in C++), but to override slots, classes and methods with each other. For example, one can decide that in a particular subclass, a slot value should in fact be computed by a function, and simply override the slot with a method.

The ability to override classes in particular has substantial repercussions, especially when coupled with Newspeak's support for nested classes. One can define an entire class library or framework as a set of classes nested within an outer class, and then modify the library via inheritance, overriding classes defined within it. This is known as *class hierarchy inheritance* [17]. The use of nested classes to encapsulate entire libraries forms the basis of Newspeak's module system. The dynamic binding of class names applies to a class' declared superclass as well, effectively making each class a mixin [6].

In the simple example above, it is possible to initialize a point's slots using its public interface. This is bad style however. It is better to initialize an object when it is created. Here is a more refined version of Point illustrating how this is done.

```
class Point x: i  y: j  =  (
(* ClassCategory:  Graphics kernel *)
(* A class representing points in 2-space *)
  public x ::= i.
  public y ::= j.
)
( (* instance side *)
(* MessageCategory: printing *)
public printString  = (
  ' x = ', x printString, ' y = ', y printString
)
```

The class declaration causes the name Point to refer to an automatically generated *class object*. Instances may only be created by invoking a factory method on Point.

Every class has a single *primary factory,* in this case x:y:. If the factory is absent, it defaults to new. The primary factory's header is declared immediately after the class name. Newspeak ensures that every instance is initialized exactly once using its primary factory. The declaration of the primary factory automatically generates an implementation. This implementation will ensure that the initialization expressions for the slots (e.g., x ::= i) are executed. The formal parameters of the primary factory are in scope in the slot initializers. We can therefore create fully initialized instances of Point e.g.,

```
Point x: 42 y: 91
```

This concludes our introduction to the Newspeak language. In the following three sections, we describe Newspeak's approach to modularity, security, and reflection. In sections 5 through 6 we shift to a discussion of the Newspeak platform, going through the GUI and IDE, and the approach to interoperability with other languages. We then discuss performance, future and related work, and our current status. Finally, section 12 concludes.

2 Modularity

In Newspeak, a top-level class constitutes a *module definition.* An instance of such a class is a *module.* Top level classes do not have access to their surrounding scope, and are inherently stateless. Newspeak has no static state: all state is always part of a specific object - an instance of a given class.

Consequently, all names that may be referenced inside a module definition must be defined by it or inherited from its superclass, which must also be a module definition. These names will include the names of the formal parameters of the module definition's factory method. While module definitions are completely stateless, modules themselves may be stateful. However, a module's only connection to the outside world comes from the actual arguments passed to the factory method that created it.

Module definitions are therefore re-entrant. Hence multiple instances of libraries can be created and deployed independently, side by side and simultaneously without interference. Such designs fall out simply and naturally, without the need to resort to complex mechanisms such as class loaders [15].

Since modules have no access to a shared global namespace, it may not be immediately obvious how a complete program is actually assembled and run. There is a need for some place to glue different modules together, by instantiating module definitions and feeding in any arguments their factories require. How is this accomplished?

The top level of a Newspeak program should be an immutable *object literal* which contains one or more module definitions, and a method main: to instantiate them. The main: method contains the glue code mentioned above. The method should take a single argument representing the underlying platform. This provides the application with its sole connection to the state of the outside world. The platform can also be used to obtain library modules that are not defined by the application source code. This is sketched below; note that the syntax of an object literal is similar to that of a class, but without the keyword **class**, the class name and the factory method signature.

```
(     ) (
     class MyApp platform: p args: args = (...)
     public main: platform = ( MyApp platform: platform args: platform commandLineArgs)
)
```

The Newspeak environment can then evaluate the object literal and invoke its main: method with an object representing the platform. This is similar to the classic C convention of invoking an application via a distinguished function main(). Our object literal is analogous to the C program, and the object literal's main: method is analogous to the main() function. However, our approach differs in a number of subtle yet crucial ways.

Our object literal is immutable, so it does not have any state itself, and no state is provided to it implicitly via global names. Instead, all state - not just the

arguments to the application - is provided explicitly via the main: method. Furthermore, any substantial application will be built from modules whose namespace is entirely self contained, and does not depend on the surrounding object.

3 Security

Newspeak is intended to provide a foundation for secure applications. As information systems become more open (e.g., allowing customers, vendors, partners selective access to data and functionality), we expect security to become a pervasive concern [4].

Newspeak's security model is founded on the object-capability model [16]. In this model, the authority to perform an operation (which may have potential security implications) is provided exclusively through objects that act as *capabilities.* This places several requirements on the programming language. These include:

1. Objects must provide true data abstraction; they must be able to hide their internals from other objects - even other objects of the same class.

2. There must be no static state (e.g., static or global variables). Such state can be accessed by code that was not explicitly authorized to do so, providing *ambient authority.*

With respect to point 1, Newspeak supports object-level encapsulation. Object members that are private or protected can only be referenced within the scope of the object. This is not the case in mainstream object-oriented languages such as C++, Java or C#.

Now consider point 2. A typical example of ambient authority might be a class File with a constructor that takes a file name and returns an instance that can access a file in the local file system. This is a standard design, but in a situation where file system access must be restricted, requires authorization checks on every access.

Systems that combine security consciousness with pervasive ambient authority, such as Java, pay an exorbitant run time cost for such checks, since they may require a traversal of the entire call stack to ensure that no unauthorized caller, however indirect, might retrieve information about a file's contents or even its existence [11]. As a result, these checks are often disabled, completely undermining security.

An alternative approach is a sandbox model, where only operations deemed safe are provided. Java also supports sandboxing via class loaders [15]. However, class loaders are complex and brittle; they can introduce interoperability problems because they create incompatible types, and they do not compose well.

As we have already noted, there is no static state in Newspeak, addressing point 2. In Newspeak, each module runs in its own sandbox, created explicitly by providing the module with the desired capabilities (i.e., objects provided as parameters to the factory when the object was constructed). There is usually no

need for explicit (and costly) security checks on individual operations to ensure that the caller has the appropriate authority to invoke them. The fact that the caller holds a reference to an object that can perform the operation conveys the necessary authoritty.

4 Reflection

Newspeak supports both *introspection* - the ability of a running program to examine its own structure - and *hotswapping*, which allows a program to modify itself while running. There is, however, a natural tension between reflective access and security. Newspeak uses a mirror based reflective architecture [8] to resolve this tension. This is key in producing a platform where intellectual property can be secured, while still leveraging the power of dynamic languages.

Whereas mainstream languages provide a uniform level of reflective access to every object via a standard method such as getClass(), in a mirror based system, reflection is mediated by objects known as *mirrors*. Mirrors serve as capabilities for reflection. Only code that is in possession of a mirror for an object can reflect on it. Different mirrors can provide varying levels of reflective access - for example, some mirrors might only allow introspection, others might restrict even introspection to the public API of a class, and others might provide unlimited access.

Mirrors can be obtained by asking an instance of class Reflection:

cm: (Reflection new reflectOnClass: c)

The code above returns a mirror on the class c, and sets the slot cm to hold it. Reflection instances have the opportunity to perform any additional authorization checks that may be required, and return a mirror object that conveys the desired reflective capabilities. In the extreme case, one could provide a Reflection class that refused to return mirrors at all, effectively disabling reflection.

We can ask the class mirror for a mirror on its mixin:

mx: cm mixin

we can then ask the mixin for its methods, slots or nested classes:

```
mtds: mx methods.
slts: mx slots.
clsss: mx classes.
```

Each of these operations returns a MirrorGroup representing the set of elements in question. Note that it is easy for a given mirror implementation to return a group that consists of some subset of the elements - for example, only the public members. The mirror group may be immutable, in which case it can be queried for specific elements, but not modified:

mtds mirrorNamed: #foo

The mirrorNamed: method looks up a method whose name matches the symbol #foo in the mtds MirrorGroup and returns a *method mirror* on it. This

provides for introspection but prevents modifications to the running system. On the other hand, a mirror group may be mutable, allowing for changes, e.g.:

```
mtds addFromSource: 'twice: x = (^2*x')
```

It is clear from the above examples that the mirror API can be refined to provide fine grained control over what reflective operations are permitted. The design of a suitable API that provides the necessary degree of functionality and control is non-trivial and ongoing.

5 GUI and IDE

5.1 GUI libraries

We have developed both a graphical widget library (called Brazil) and an application framework (known as Hopscotch [9]). Brazil supports the development of UI code that is independent of the underlying platform. Brazil code can be *dynamically* bound to an underlying native GUI.

The initial implementation of Brazil used the Morphic GUI library provided by Squeak Smalltalk as its sole binding. We have since developed a Windows native binding which allows applications to display their GUI using native widgets when they are run on Windows. Native bindings for Unix will be introduced in the second half of 2008.

Hopscotch is a general purpose GUI application level framework built upon Brazil. Application developers rarely need to use Brazil directly; they usually work with Hopscotch. Interaction with Hopscotch combines desirable features of web browser style navigation with support for hierarchical decomposition.

Hopscotch is built upon the notion of *presenters*, which are extensible and customizable objects designed to present a specific kind of data. Presenters are created using *presenter combinators*, which allow for the hierarchical composition of presenters. The combinators impose a navigational paradigm that is similar to the one used in web browsers. This allows for easy navigation of graphs of presenters. Unlike web browsers, presenters can be hierarchical, so Hopscotch applications are well suited to displaying hierarchical data (e.g., tree structures such as file systems or program code). The combination of navigation and hierarchy imposed by Hopscotch's combinators naturally handles the display of any hypergraph of user data.

5.2 IDE

Initially, we relied on the Squeak IDE, extending and adapting it to work with Newspeak. The concept of an IDE originates with Smalltalk, and like most Smalltalk implementations, Squeak provides a rich and powerful development environment.

Nevertheless, as Newspeak has evolved, we found that the existing Smalltalk tools were not always well suited to working with Newspeak. Newspeak programs make extensive use of class nesting, and the traditional Smaltalk class

browsers do not support this very well.

In response, we have developed an IDE for Newspeak based on the Hopscotch framework. The IDE supports class browsing, object inspection, interactive evaluation, and source control management. As of this writing, work on a Hopscotch-integrated debugger is in progress; until it is complete, we rely on the existing Squeak debugger, which we have adapted to deal with Newspeak code.

Figure 1 shows a screen shot of the Hopscotch IDE, displaying a class browser on a Newspeak module definition. The module shown implements a parser combinator library [5]. It contains a considerable number of nested classes. Each nested class can be browsed in several ways:

- By following a link, replacing the display of the module definition with a class browser on the selected class.
- By expanding it in place, presenting a nested class browser in the context of the surrounding class (e.g., the class SequentialParser in the figure).
- By opening the link in a new browser window.

Developing the Hopscotch IDE was useful both for testing the efficacy of Hopscotch and for improving the already superb development facilities available to Newspeak developers. The Hopscotch paradigm works very well in the IDE context; it naturally provides seamless integration among an open-ended set of tools, which is a defining characteristic of a good IDE.

We find the Hopscotch based IDE extremely effective and pleasant to use. Traditional UI design tends to create displays which include a large number of panels, toolbars, sidebars etc., which tend to overwhelm users with complexity. An alternative is to produce a UI with many windows, which also become burdensome. Hopscotch avoids this dilemma. One rarely needs to open more than one Hopscotch window, yet the display in a given window is typically simple and manageable.

The benefits of using Hopscotch are not, of course, limited to the IDE. Hopscotch was designed as a general purpose application framework. The same ease of learning (due to the similarity to web browsers and file browsers) and ease of navigation carry through to the application GUI as well.

6 Interoperability and FFI

A key requirement for any new programming language is that it provide efficient and easy to use facilities for interacting with programs running in the broader environment. This is often challenging for a high-level language, as there is naturally an impedance mismatch between the high level constructs the language provides and the abstractions of the underlying system.

Newspeak includes facilities for calling out to other programming languages, and being called by them. The generic term for such facilities is a *foreign*

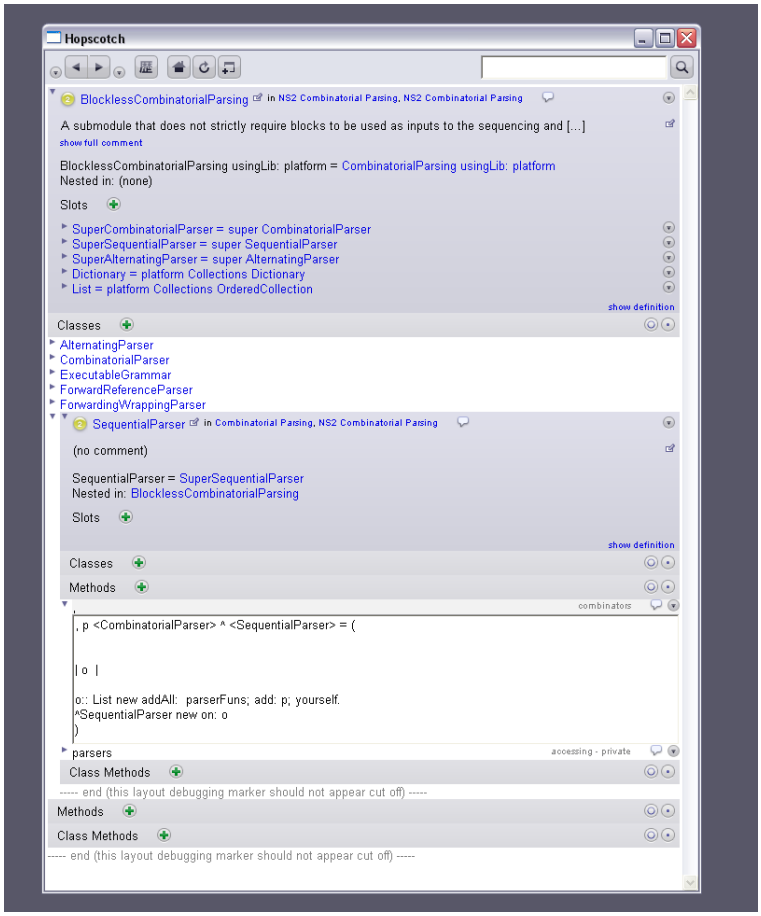

Figure 1: Screen shot of Newspeak's Hopscotch browser.

function interface (FFI). Since Squeak did not provide a reliable, performant and easy to use FFI, we have implemented one ourselves.

The first use of the FFI has been the implementation of the native GUI bindings for Brazil. The nature of GUI systems requires both the ability to call out to the native GUI, and the ability for the GUI to call back in to Newspeak (when relevant UI events must be handled).

We expect Newspeak applications to integrate closely with non-Newspeak applications at Cadence and elsewhere, which will also require FFI support. Applications written in other languages should be able to view a Newspeak application via a DLL and associated C header files. These can be generated automatically utilizing the FFIs facilities for handling call-backs.

From within Newspeak, foreign calls are method invocations on suitable proxy objects known as *aliens*, which represent constructs in another language (e.g., C functions). These are produced by the base layer of the FFI. Currently, the proxies are created by manually calling this base layer. Higher levels of the API are under development; these will allow a programmer to specify the relevant DLLs and programming language declarations (e.g., .h files) that describe and implement the desired foreign language functionality, and automatically produce the desired proxies. Moreover, a Newspeak object representing the foreign API as whole will also be produced, allowing the programmer to interact with foreign code at a high level of abstraction.

It is important to note that the high level API is made possible by two characteristics of Newspeak: its metaprogramming facilities, which in turn are largely enabled by dynamic typing; and the uniform abstraction of objects, which can naturally represent entire APIs and their constituents.

Our FFI architecture allows for interaction with an open-ended set of foreign languages of varying characteristics. The most critical need is always to interact with the application binary interface (ABI) of the underlying operating system, which is typically written in C. We have implemented a C FFI on both Windows and Unix. We have also validated that our approach works for other languages by constructing an Objective-C alien interface as well. We are confident that alien interfaces for Java or C# can be constructed as necessary.

Crossing language boundaries inevitably entails some performance overhead due to marshaling and unmarshaling of arguments, data type conversions etc. However, we find the overhead of our FFI implementation entirely acceptable. A foreign call can be accomplished in a few microseconds.

We believe our alien based approach addresses all the key requirements of an FFI. It provides for calling out of, and in to, Newspeak; It has proven itself to work reliably; Its performance is adequate; and it leverages the uniform use of objects throughout Newspeak to provide a natural and easy to use interface for the programmer.

7 Performance

Our implementation is based upon the open source Squeak Smalltalk virtual machine. Smalltalk virtual machines are, as a rule, significantly faster than implementations of scripting languages at a comparable level, such as Javascript, Ruby, Perl or Python. Many scripting languages rely heavily on subsystems implemented in C to provide adequate performance in specific areas of specialization, such as string and regular expression processing, but provide poor performance on general purpose code.

Consequently, even though Squeak is not a high performance implementation of Smalltalk, it has provided us with adequate performance so far, with very little implementation effort.

Newspeak is a very high level language, and could certainly benefit from a highly tuned implementation. High performance open-source virtual machines such as Strongtalk [1] or Self could be used to run Newspeak, though considerable effort would be needed to adapt them and port them to the relevant platforms of interest. Another promising direction is the growing trend on mainstream platforms to support dynamic languages.

Newspeak performance is an order of magnitude better than current Ruby or Javascript implementations. While current raw performance is no doubt significantly slower than a high quality mainstream language implementation such as Java or C#, observed response time for an interactive client application is attractive, and in fact superior to Java, due to better start up time and reduced footprint. Based upon our experience with Strongtalk, we believe that Newspeak peak performance can be driven to approximately 50-70% of mainstream technologies.

8 Future Work

Major areas of work remain until Newspeak realizes its full potential. Many of the subsystems described above are still maturing: our native GUI bindings, higher level alien interfaces for additional languages, improved performance etc.

A major ongoing focus of our work is providing platform support for synchronizing both data and programs between Newspeak processes (e.g., clients and servers) as proposed in [2].

With the growing importance of web browsers as a platform, a web based implementation targeting Javascript is of significant interest. This requires a fully bootstrapped version of the platform, so that code developed on the Squeak version can operate on the web and vice versa. However, at the moment Newspeak is heavily dependent on the host Smalltalk environment and its core libraries, such as collections, streams, numerics etc. Our goal is to make Newspeak stand on its own, and so Newspeak versions of these libraries are necessary. Because

of the similarities between Newspeak and Smalltalk, we believe open source versions of these libraries can be adapted with relatively small effort.

We expect to support actor-based concurrency, informed by the principle that shared state and concurrency are a dangerous mix unsuitable for general purpose application development. We also intend to extend Newspeak with an optional type system and a framework for pluggable types [3], enabling the language to enjoy most benefits of static type systems without incurring their limitations.

9 Related work

Newspeak is a direct descendent of Smalltalk [10]. Like its predecessor, Newspeak is purely object-oriented, meaning that all data, even basic types such as characters and integers, are objects that communicate exclusively via virtual method calls. However, in Smalltalk, internal computation within an object may access state directly. This means that Smalltalk code within a class and its subclasses is not representation independent.

In contrast, Newspeak follows the lead of the Self [19] programming language in insisting that all computation is late bound. As a result, programs enjoy the property of representation independence. Unlike Self, Newspeak is a class based language. We believe this gives the language improved modeling capabilities, and makes it more accessible to practitioners. It also makes it easier to implement the language efficiently.

Newspeak supports nested classes, a notion that originates in the language Beta [14]. Unlike nested classes in Java, nested classes in Newspeak may be overridden in subclasses.

Newspeak's object capability model is influenced by the language E [16]. Unlike E, Newspeak provides full support for reflection, including both introspection and hotswapping.

10 Experience

We have made direct use of the innovative language features listed above, such as mixins and class hierarchy inheritance, and have found them extremely effective. An example is our parsing library [5], where we have found the combination of closures, class hierarchy inheritance, mixins, the ability to override slots with methods, dynamic typing and Newspeak syntax to be extremely expressive.

Measuring programming productivity is difficult, but the fact that a team of 4 people has been able to construct the system in about a year speaks for itself.

11 Status

The design of Newspeak began in late October of 2006. Since then, the Newspeak team has grown to a total of 4 developers.

Newspeak is currently being used by small application team within Cadence. Their experience has been overwhelmingly positive. As we gain more experience, and the language matures, we expect the platform to be useful for other projects within Cadence and beyond.

Newspeak is far from complete. When we feel the system has progressed sufficiently, we plan to release Newspeak as open source software.

12 Conclusion

Newspeak draws on ten years of experience maintaining and extending the Java programming language [15, 12, 13, 18]. It is the first class based language that consistently applies the principle that all names act as dynamically dispatched method invocations. This applies not only to methods, but also to slots (representing storage locations) and classes as well.

As a result, Newspeak pervasively enforces the concept of programming to interfaces rather than implementations, making code representation independent. Newspeak has a powerful module system, and supports class nesting, mixins and class hierarchy inheritance.

The language supports the object capability model of security, in particular completely eliminating static state, ensuring that Newspeak modules are always re-entrant. The object-capability model, module system, mirror based reflective architecture, pluggable types and message based programming synergize to provide a very powerful framework that is extremely flexible, yet robust enough for secure, well engineered mission critical software.

References

[1] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Strongtalk website. http://www.cs.ucsb.edu/projects/strongtalk/.

[2] G. Bracha. Objects as software services. Invited talk at OOPSLA 2005 Dynamic Languages Symposium; updated video available at http://video.google.com/videoplay?docid=-162051834912297779. Unpublished manuscript available at http://bracha.org/objectsAsSoftwareServices.pdf.

[3] G. Bracha. Pluggable type systems, Oct. 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[4] G. Bracha. Toward secure systems programming languages, Mar. 2004. Keynote address. Slides available at http://www.acm.org/conferences/sac/sac2004/keynote_speakers.htm.

[5] G. Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, Oct. 1990.

[7] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Sept. 1993.

[8] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 2004.

[9] V. Bykov. Hopscotch: Towards user interface composition, July 2008. Submitted to ECOOP 2008 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT).

[10] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[11] L. Gong, G. Ellison, and M. Dageforde. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, Reading, Massachusetts, 2000.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.

[14] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The Beta Programming Language. In *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[15] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*.

[16] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[17] H. Ossher and W. Harrison. Combination of inheritance hierarchies. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 25–40, Oct. 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[18] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97116, December 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, http://www.jot.fm/issues/issue_2004_12/article5.

[19] D. Ungar and R. Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 1987.