# Why Smalltalk Wins the Host Languages Shootout

Lukas Renggli
renggli@iam.unibe.ch

Tudor Gîrba
girba@iam.unibe.ch

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

## ABSTRACT

Integration of multiple languages into each other and into an existing development environment is a difficult task. As a consequence, developers often end up using only internal DSLs that strictly rely on the constraints imposed by the host language. Infrastructures do exist to mix languages, but they often do it at the price of losing the development tools of the host language. Instead of inventing a completely new infrastructure, our solution is to integrate new languages deeply into the existing host environment and reuse the infrastructure offered by it. In this paper we show why Smalltalk is the best practical choice for such a host language.

## Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Domain engineering; D.3.2 [**Programming languages**]: Smalltalk

## General Terms

Design, Languages

## Keywords

Embedded Languages, Domain-Specific Languages, Programming Environments and Tools

## 1. INTRODUCTION

With the increasing demand to combine multiple languages within a single project, different solutions have been proposed to simplify the process of building and using polyglot programming environments. While these solutions have their strengths at various levels, they do not cover the complete spectrum of integrating these languages and of offering development tools for them.

We use the term *host language* to refer to the language that is used as the basis for implementing new languages and for gluing them together. Furthermore, we define *context specific languages* as languages that are embedded in a host language, but active only within certain well-defined contexts.

As a running example we use the Extended Backus-Naur Form [15], as a simple language extension to an existing host language. The possibility to use the EBNF directly within the code of the host language raises the conciseness of a parser definition considerably. An example grammar to parse numbers might look like this:

```
digit = "0" | "1" | ... | "9" ;
number = [ "−" ] digit { digit } [ "." digit { digit } ] ;
```

This language would have to coexist with the host and possibly with other languages. This co-habitation should be transparent in the sense that objects can be passed through code written in multiple languages. Furthermore, ideally the environment should provide development tools, like syntax highlighting and debugging, that can be used uniformly across languages.

In this paper we evaluate seven general purpose languages (C++, C#, Java, Javascript, Lisp, Ruby, and Smalltalk) from the point of view of the mechanisms they offer for language integration. The shutout is performed as follows: we first identify the requirements for a host environment, we distill the features of a programming language that would support these requirements and we compare the considered languages. From our comparison, Smalltalk wins the shutout.

The paper is structured as follows: Section 2 details the requirements for a host environment and compares with related work. Section 3 gives a quick introduction to the Helvetia system. Our main contribution is presented in Section 4 where we discuss the advantages and disadvantages of using Smalltalk as the host environment. Section 5 concludes the paper.

## 2. REQUIREMENTS FOR A HOST ENVIRONMENT

We identify three major features that are generally needed to support the embedding and combining of multiple languages into a single host environment:

**Multiple context specific languages.** Different languages and the host language should be mixable in arbitrary ways. Language changes should not be limited to file boundaries, but should depend on the location in the source code only. In the example of the EBNF language extension, we would like to define the grammar close together with the associated production actions that are specified using the host language syntax.

**Homogeneous language integration.** It should be possible to pass values from one language to another without requiring a conversion in-between. Similar transparent interaction between the meta-level where a language is defined and the base-level where a language is applied should be possible. Homogeneous language integration [11] enables all languages to be aware of each other and make use of the common reflective facilities of the host system to reason about themselves. In our running example we would like to directly access and use the grammar and the resulting tokens from the host language.

**Homogeneous tool integration.** Language users demand sophisticated tool support for the languages they are using. For example, they would like to step with a single debugger through a method that mixes various languages. To debug a grammar definition, we would like to be able to step both through EBNF and through the production actions using the debugger of the host environment.

The most basic approach is to derive a new pseudo-language from an existing API. This technique is known as a *Fluent Interface*, a form of an *internal DSL*. While this approach fulfills all the above properties, it is often not powerful enough as the language is constrained by the syntax and the semantics of the host environment. For example, instead of using the concise EBNF language constructs we would need to express grammars using a verbose series of message sends written in the host language.

Systems with *meta-programming* facilities like Scheme, Converge [14] or MetaOCaml [5] avoid that problem by providing compile time code generation, however they often lack sophisticated tool support.

Similarly, *extensible compilers* like JastAdd [9] or Xoc [6] allow language designers to tweak the host language compiler, but usually don't provide a way to integrate the modified language into the existing tools. None of the systems offers tight IDE integration, and the transformed code cannot be debugged at the source level.

*Language workbenches* like JetBrain MPS [8] or Intentional Software [12] come with a specialized IDE for language engineering. They provide a special workflow to define new languages and they provide tools for language development and application. The problem with these approaches is that they do not build on top of existing tools and host languages, but instead provide their own custom toolset.

We implemented a host environment, Helvetia, using a different strategy. We chose a host environment and we extended the existing compiler and programming environment to allow us to parameterize them for language extensions. To support our approach the host environment needs to support the following six features:

1. A *minimal syntax* makes a language a good source and target for program transformation.

2. *Dynamic semantics* allows language designers to change and extend the behavior of existing classes. Furthermore, dynamic typing lets developers replace objects as long as the replacement understands the expected messages.

3. *Reflective facilities* makes the structure and behavior of a system observable and changeable. This is crucial for tools as well as the language extensions themselves.

4. A *homogeneous language* is a language that is implemented in itself, thus is specifically easy to extend and change.

5. *Homogeneous tools* are tools that are written in the host language itself. Again this makes them viable for change.

6. Being able to change a language *on the fly* makes the development process faster and quick language experiments feasible.

After careful consideration we chose Smalltalk as the host language for Helvetia. Helvetia covers all above mentioned requirements and transparently blends into Smalltalk and its development tools. While the abstract approach is not limited to Smalltalk, the choice of Smalltalk did present several practical benefits over other solutions. This paper distills our experience and presents the arguments for why Smalltalk is the best choice.

## 3. HELVETIA IN A NUTSHELL

In this section we present Helvetia[1] shortly. The goal of this description is not necessarily to describe Helvetia completely, but rather to provide the necessary background from a user perspective.

Helvetia is an extensible framework that enables language designers to cleanly extend compiler and development tools of the standard Smalltalk IDE. As depicted in Figure 1, our approach reuses the existing tool-chain of editor, parser, compiler and debugger by leveraging the abstract syntax tree (AST) of the host environment. Different languages cleanly blend into each other and into existing code. The same tools can be reused with different language extensions.

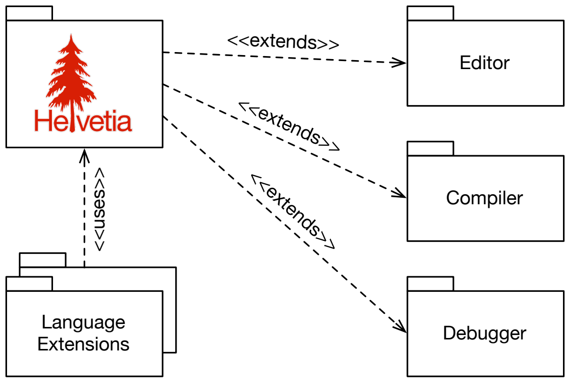


**Figure 1: The Helvetia System.**

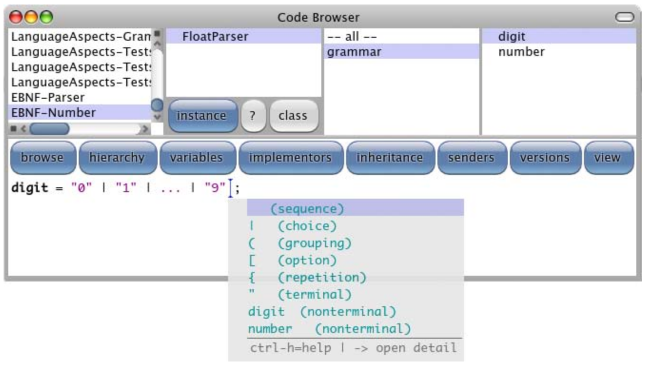
The Helvetia framework is lightweight in the sense that it is implemented in less than 900 lines of code. Helvetia makes heavy use of libraries that are part of the Smalltalk system:

[1]The implementation along with its source code and examples can be downloaded from http://scg.unibe.ch/research/helvetia.

- The *Refactoring Engine* [10] is central to any Smalltalk system. HELVETIA mostly makes use of its rewrite engine to declarative specify transformations of the abstract syntax tree (AST).

- The *New Compiler* is an extensible compiler built on top of the AST of the refactoring engine. It transforms ASTs to bytecodes that can be directly executed by the virtual machine (VM).

- *SmaCC* [4] is an LALR based parser generator framework used for example by the *New Compiler* to parse Smalltalk source code. *PetitParser* is a lightweight alternative based on parsing expression grammars (PEG).



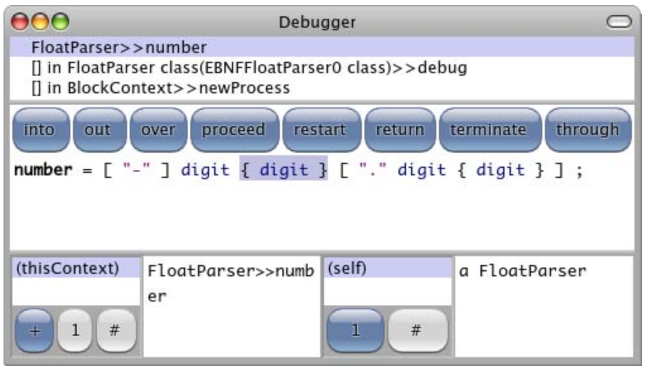
```
digit = "0" | "1" | ... | "9";
```

**Figure 2: The Smalltalk browser opened on the EBNF language with adapted syntax highlighting and auto completion.**

Furthermore, HELVETIA uses and extends tools:

- The *OmniBrowser* framework is a toolkit to build extensible development tools, see Figure 2. HELVETIA extends the standard code browsers and debuggers with custom functionality, such as contextual language specific menu actions.

- *Shout* and *eCompletion* are the standard plugins for syntax highlighting and auto completion. We extended these tools to be able to use them on arbitrary languages.



```
FloatParser>>number
[] in FloatParser class(EBNFFloatParser0 class)>>debug
[] in BlockContext>>newProcess
```

```
number = [ "-" ] digit { digit } [ "." digit { digit } ] ;
```

```
FloatParser>>number
a FloatParser
```

**Figure 3: Stepping through a mixture of EBNF and the host language using the standard debugger.**

Figure 3 shows how we step through a custom language to define grammars using the traditional Smalltalk debugger. The debugger displays the original source-code and properly highlights the current execution location, even though the code has been transformed to a standard Smalltalk AST to get into an executable state. Language specific syntax highlighting is provided as in any other editor.

## 4. WHY SMALLTALK

In this section we present the case for why Smalltalk is an optimal solution as host environment in comparison to other programming languages. Table 1 provides a summary of the features supported by the considered programming languages: a filled circle denotes that the language supports fully the given feature, while a half-filled circle means that the feature is only partially supported or that it requires additional workarounds to access it. Each of the features is presented in detail in the following subsections.

### 4.1 Minimal Syntax

Smalltalk has a minimal syntax[2], and Smalltalk compilers rarely have more than ten different node types to support the full language. Depending on the implementation details, the following node types are supported:

1. A *method node* is used to describe the method signature and method body.

2. A *sequence node* is used to describe a sequence of statements and a preceding declaration of temporary variables.

3. A *message send node* is used to describe a method invocation on a receiver with a given set of arguments.

4. A *cascade node* describes a series of message sends to the same receiver.

5. A *block node* describes a block closure and its arguments.

6. A *return node* is used to describe a return from a method or block.

7. A *variable node* describes a temporary, instance or global variable reference.

8. An *assignment node* describes a variable assignment.

9. A *literal node* describes literal values, such as numbers, characters, strings, symbols or boolean values.

The rest of the language features come from the Smalltalk library. Contrary to most other programming languages, control structures are modelled using message sends and block closures, thus the compiler does not require specific node types to handle these.

The simplicity of Smalltalk makes it a very attractive target for language transformation both from arbitrary languages to Smalltalk or within the Smalltalk language itself. In the first case a parser can directly build a Smalltalk AST, in simple cases just consisting of a series of message sends.

[2]Jokingly it is often remarked that a description of the syntax would fit on a business card.

|  | C++ | C# | Java | Javascript | Lisp | Ruby | Smalltalk |
|---|---|---|---|---|---|---|---|
| 4.1 Minimal Syntax | ○ | ○ | ○ | ○ | ● | ○ | ● |
| 4.2 Dynamic Semantics | ○ | ◐ | ○ | ● | ● | ● | ● |
| 4.3 Reflective Facilities | ◐ | ◐ | ◐ | ◐ | ● | ◐ | ● |
| 4.4 Homogeneous Languages | ○ | ○ | ○ | ○ | ● | ○ | ● |
| 4.5 Homogeneous Tools | ○ | ○ | ◐ | ◐ | ◐ | ○ | ● |
| 4.6 On-the-fly Programming | ○ | ○ | ○ | ◐ | ◐ | ◐ | ● |

**Table 1: Comparison of different main-stream programming languages and their suitability for language engineering. Legend: ○ no support, ◐ partial support, ● full support.**

Transformations within the language only need to match a few basic cases to cover the complete language specification.

In the example of the EBNF language we transform the input into a series of message sends that construct an object model of the grammar. The example grammar presented in Section 1 is transformed to the AST of the following two Smalltalk methods:

```
digit
    ^ $0 asParser to: $9 asParser

number
    ^ $− asParser optional , self digit plus , ($. asParser , self
       digit plus) optional
```

The only contenders in this area are Lisp-like languages. This family of programming languages provides s-expressions (parenthesized lists) as their central language construct. This means that source code is written in an extremely uniform way that is directly related to the abstract syntax tree. As such Lisp is very well suited for macro programming. On the other hand, the syntax of Smalltalk is close to natural language, and thus targeted at readability while still being simple enough for transformations.

## 4.2 Dynamic Semantics

Smalltalk is built around objects, polymorphism and dynamic dispatch. This together with the fact that everything is a message send is an advantage when it comes to changing the semantics. For example, to change the default lower index of arrays of 1 to something else, is simply a matter of creating a custom subclass of Array and overriding the methods at: to read and at:put: to write an array cell.

In the example of the EBNF language we extended the classes of common Smalltalk objects with the message asParser , so that these objects can be converted to a parsers that accept themselves. This is used in the transformed code to construct a parser for a character. $0 asParser returns a parser that parses the character 0, see the listing in Section 4.1.

However even though it is advertised that everything in Smalltalk is a message send, this is not entirely true. For example, reading from and writing to temporary, instance and global variables is not performed using a message send, but through primitive bytecodes.

Bracha *et al.* [2] have demonstrated with *NewSpeak* that we can build a Smalltalk-like system that accesses state through message sends only. This presents the advantage that state access can be overridden and intercepted as it is currently done with method polymorphism. Intercepting state changes is useful to automatically notify observers that are interested in how a particular object changes.

Most programming languages today provide static built-in types that have fixed semantics and that cannot be changed. Furthermore, it is often not possible to extend the existing system or library classes with new code (*e.g.,* Java). C# provides an extension mechanism through partial classes, however this mechanism does not allow us to extend existing tools as the partial class and its extensions must reside in the same module. In dynamic languages like Ruby and Javascript it is typically possible to extend existing classes with new methods like Smalltalk does.

## 4.3 Reflective Facilities

The Helvetia system heavily depends on the reflective features of the host language. We use the reflective infrastructure to scope language extensions to classes, class hierarchies, packages, *etc.* The EBNF language extension if for example scoped to the subclasses of a generic parser class.

Furthermore the transformation of the parse trees are performed using the rewrite tools of the Smalltalk refactoring engine. Instead of relying on string transformations or code generation, we transform the language extension AST into the Smalltalk AST and we directly pass it to the compiler. This approach allows us to keep accurate source location information, which is crucial to facilitate contextual error reporting and highlighting in the debugger.

However, while Smalltalk has an excellent infrastructure for reflection, it lacks features that are central to metaprogramming. Traditionally new code fragments are specified using strings and string concatenation. This leads to fragile code and makes it difficult to debug, as the origin of the code cannot be tracked. A slightly better solution is to manually instantiate and compose the AST nodes. In this case the origin can be tracked, but the code is still hard to read and debug.

*Quasiquoting* facilities known from Lisp [1] or OMetaCaml's staging constructs [13] promise rescue. We extended the Smalltalk language with an expressive quasiquoting infrastructure. We introduce the following operators, that can be used as a prefix for any Smalltalk expression:

- **Quasiquote.** An expression prefixed with `` is delayed in execution and represents the AST of the enclosed expression at runtime.

- **Unquote.** An expression prefixed with `, can be used within a quasiquoted expression. It is executed when the AST is built can be used to combine smaller quasiquoted values to larger ones.
- **Splice.** An expression prefixed with `@ is evaluated at compile-time and the result is spliced-into the code. If the returned expression is not an AST, it is automatically lifted to the AST level, *e.g.,* by introducing a literal node.

As an example we use these operators to generate code to calculate $x^n$, where $n$ is a positive integer. The method below is a recursive definition of this method written in regular Smalltalk:

```
raise: x to: n
  ^ n = 1
    ifTrue: [ x ]
    ifFalse: [ (self raise: x to: n − 1) ∗ x ]
```

If we want to avoid the recursion at runtime and instead generate code that directly calculates the result for a given integer $n$ we annotate the code with quasiquote and unquote operators:

```
raise: aNode to: n
  ^ n = 1
    ifTrue: [ aNode ]
    ifFalse: [ ``(`,(self raise: aNode to: n − 1) ∗ `,aNode) ]
```

When evaluating self raise: ``x to: 3 with a variable node ``x, a parse tree is constructed that multiplies the variable x three times with itself yielding x ∗ x ∗ x. Using the splice operator we can insert the generated parse tree anywhere into the source code. For example:

```
qubic: x
  ^ `@(self raise: x to: 3)
```

This creates code equivalent to:

```
qubic: x
  ^ x ∗ x ∗ x
```

In our running example, the quasiquoting facilities simplifies the code transformation from the EBNF to the host language. The three examples below show different approaches to generate a small part of the code we saw in action in Figure 3. Specifically we show how the repeat statement is composed:

*1. String Concatenation..*
The most trivial way to do this is to (1) print out the inner node, (2) concatenate it with the repeat message that is part of the API of the language grammar model and returns a repeat clause, and (3) then re-parse the complete string. Code like this is hard to debug and with pretty printing and parsing origin information is lost. Furthermore, repeatedly parsing and pretty printing code is also very inefficient.

```
Parser parseExpression: '(' , aNode prettyPrinted , ') repeat'
```

*2. Manual AST Composition..*
Another possibility consists to manually construct the AST. In this case the node is composed with the repeat message. This approach works reasonably well, but it gets cumbersome with more complicated examples. The compiler cannot check up front if the resulting code is valid and it is not immediately obvious for developers to see what code gets generated.

```
RBMessageNode receiver: aNode selector: #repeat
```

*3. Quasiquoting..*
Using the introduced quasiquoting facilities code is easily generated. Furthermore, it is immediately visible what kind of code is generated and the compiler can validate the code generation in advance.

```
``(`,aNode repeat)
```

The presented quasiquoting language extension to Smalltalk is simple and does not conflict with the existing syntax. We strongly encourage the Smalltalk ANSI committee to include quasiquoting in a future standard proposal. The fact that Smalltalk entirely lacks sophisticated facilities for meta-programming could be fixed by implementing quasiquoting as a language extension.

Unfortunately only very few mainstream programming languages (*e.g.,* Javascript) provide rich structural and computational reflection. Furthermore, even fewer provide support that goes beyond basic structural reflection at the level of classes or methods. C# 3.0 provides only partial access to the AST of statically declared expressions using expression trees. Only in Lisp and Smalltalk we do have direct access to the AST. Although Helvetia does not strictly require reflective facilities to change the running application, having read-write access to the AST greatly simplified its implementation.

For a detailled comparison of the reflective features in different programming languages we refer the reader to the work of Bracha *et al.* [3].

## 4.4 Homogeneous Languages

Smalltalk being implemented in itself, makes it a viable target for language experiments. Although Smalltalk does not come with a fully extensible compiler, this can be easily added by introducing hook methods into the standard compiler framework that is itself implemented in Smalltalk. We have done so as described in Section 3.

In Smalltalk classes can define a custom parser and compiler by overriding the method compilerClass. Helvetia does so by overriding this method in Object, the root of the class hierarchy. This enables Helvetia to return a more sophisticated facade object that scopes language changes even further, not only at the level of classes, but also at the level of methods and at the sub-method level [7]. As the parser, the compiler and the executable bytecode are fully accessible using the reflective environment, any part of the system can be customized, extended or even replaced.

Furthermore, since all executable code eventually ends up in a compiled method object that the VM knows how to interpret, any code can be invoked without knowing its origin. As the object model is the one of the host system, objects can be transparently passed around and used by different language extensions. Thus, different languages can live homogeneously next to each other and interact in a natural and transparent way.

For example, our EBNF language would just return a series of parse tokens by default. To attach production actions to the grammar we need to be able to intermix the EBNF with normal Smalltalk code. In the excerpt below we show that we can use normal Smalltalk code to define a production action right after the grammar specification. In this case aToken implicitly refers to the character consumed. We use normal Smalltalk code to convert this character into a number:

```
digit = "0" | "1" | ... | "9" ;
  aToken asciiValue − $0 asciiValue
```

Language extensions are scoped to certain parts of the system (*e.g.,* specific classes or packages). When using the reflective facilities of the host system, different languages are aware of each other and can be closely integrated.

None of today's popular programming languages provide out of the box support for the use of different parsers and compilers. Thus people have to use a source-to-source transformation in a pre-compilation phase, or rely on a custom compiler. This leads to various problems: (1) the interaction between different languages is difficult, (2) incompatibilities exist between the custom AST representations and the domain models involved, and (3) it is often not possible to trace easily the transformed code back to the original source.

Another language besides Smalltalk that provides homogeneous language is Lisp. In Lisp, *reader macros* are used to read and transform the source code to s-expressions. Common Lisp comes with a set of reader macros that define the standard language, and custom ones can be added by developers to extend and change the syntax of the host language. The system knows about all the active reader macros and uses s-expressions as the common representation of data.

### 4.5 Homogeneous Tools

Arguments similar to those given in the previous section can also be given in relation to tools integration. All Smalltalk development tools are implemented themselves in Smalltalk and can be modified on the fly. This makes it easy for building and integrating languages into these tools. Since the tools rely on the reflection facilities, many parts of the editors can be changed just by providing different answers to their queries. For example:

- *Syntax highlighting* (see Figure 2) is typically implemented by traversing the parse tree of the edited method. As long as this tree can be properly visited by the syntax highlighter, the code editors do not care about the language that is being edited. The only information a language extension needs to provide is some color and style information so that the parse tree tokens can be highlighted accordingly.

- *Code completion* (see Figure 2) typically works on the parse tree. Language extensions are able to provide possible completion tokens that are presented to the developer.

- *Code debugging* (see Figure 3) works at the bytecode level. To highlight the current execution position in the source code, the debugger uses a source map provided by the compiler that encodes text ranges to bytecodes. By providing a custom source map, it is possible to accurately step through a mixture of different languages with a single debugger. The debugger interprets the bytecodes and uses the source map regardless of how the language looks like to the developers.

Eclipse, NetBeans and IntelliJ IDEA are full featured Java IDEs implemented in Java. As such, these IDEs provide homogeneous tools that can be extended through an expressive plugin architecture. However, developers are restricted to the provided interface and are often required to restart the complete IDE when a plugin changes. LispWorks is an IDE for Lisp development resembling Smalltalk IDEs. While it provides a rich API to extend its tools, the source code is not available and thus the developer is restricted to the provided extension points.

Having the live source code of all tools at hand is a big advantage for efficient language development and integration. In Smalltalk the compiler, editor, debugger, *etc.* can be changed, adapted or extended without limiting the developer to a plugin architecture imposed by the vendor.

### 4.6 On-the-fly Programming

The image encapsulates the running Smalltalk system. It includes all objects, all classes and their source code, and the currently executed threads. An image can be saved to the file-system at any time and in any state, and re-run on a different machine. When working in a Smalltalk system, code is compiled and installed into the running system. The typical edit-compile-run cycle is avoided, as soon as the source code is edited, it is automatically compiled and used by the running system.

Having an ever running system makes it viable to quickly develop and test new language features in the context of a domain. The language change is immediately available and can be tested in the running system using the objects already present.

When a language definition changes, it is often required that the users of this language are recompiled. In a reflective system like Smalltalk the clients of a language can be enumerated and asked to recompile themselves. This is a similar query to the functionality of displaying senders and implementors of a particular method selector.

While many dynamic languages (*e.g.,* Lisp, Ruby, Javascript) provide similar functionality through their interactive consoles, they do not take it as far as Smalltalk does. For example, it is often not possible to fix a bug from within the debugger, or to change the way the console works while it is running. The fact that source code primarily lives in files, makes it hard to interact with the code using a first-class representation.

Smalltalk being an ever living object space presents also presents practical disadvantages, as it makes it difficult to make changes in certain parts of the system, *e.g.,* changing the compiler while it is being used to compile its own source code. To circumvent these types of problems, in practice we always keep the original compiler around so that it can replace the default compiler in case something goes wrong.

Another related problem is the fact that language extensions need to be available before any of the client code is loaded. This enforces that language extensions are packaged and loaded separately beforehand.

## 5. CONCLUSION

Context specific languages are languages that are embedded in a host language, but active only within certain well-defined contexts. Embedding such new languages into an existing host environment is currently not well supported. Instead, to accommodate them we need to extend an existing language with a proper environment.

We built such an environment by expressing foreign languages in terms of the AST of the host language. This is the shortest path to reusing the host language tools, as they all work on the standard reflective facilities of the host language's code model.

The contribution of this paper is to distill our experience of using Smalltalk as the host language. We considered multiple language environments from the point of view of their suitability as possible hosts. In essence, we argue that Smalltalk is a prime candidate for a system like HELVETIA. Other languages considered (as seen in Table 1) fall short from various points of view. Lisp is a strong contender, however it lacks support of having full access to compiler and tools in the running system.

While Smalltalk is a good practical solution, it still is not ideal. To easily specify code transformation we had to extend the language with a quasiquoting mechanism. Another problem is that Smalltalk does not give us access to the execution semantics of the VM. Accommodating a language that is not message-based (*e.g.,* Prolog or Haskell) is difficult and requires mapping the semantics of the new language [16] to the message-based one of the Smalltalk VM.

## Acknowledgments

We thank Oscar Nierstrasz for his feedback on earlier drafts of this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

## 6. REFERENCES

[1] A. Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

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

[3] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[4] J. Brant and D. Roberts. SmaCC, a Smalltalk Compiler-Compiler. http://www.refactory.com/Software/SmaCC/.

[5] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, GenSym, and Reflection. In *In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science*, pages 57–76. Springer-Verlag, 2003.

[6] R. Cox, T. Bergan, A. T. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. *SIGARCH Comput. Archit. News*, 36(1):244–254, 2008.

[7] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, Oct. 2007.

[8] S. Dimitriev. Language oriented programming: The next programming paradigm. *onBoard Online Magazine*, 1(1), Nov. 2004.

[9] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–18, New York, NY, USA, 2007. ACM Press.

[10] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[11] T. Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[12] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[13] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[14] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM TOPLAS*, 30(6):1–40, 2008.

[15] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[16] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.