Language Boxes: Bending the Host Language with Modular Language Changes.

Lukas Renggli, Marcus Denker, Oscar Nierstrasz

Language Boxes*

Bending the Host Language with Modular Language Changes

Lukas Renggli[1], Marcus Denker[2], and Oscar Nierstrasz[1]

[1] Software Composition Group, University of Bern, Switzerland
{renggli,oscar}@iam.unibe.ch
[2] PLEIAD, University of Chile, Santiago, Chile
denker@acm.org

Abstract. As domain-specific modeling begins to attract widespread acceptance, pressure is increasing for the development of new domain-specific languages. Unfortunately these DSLs typically conflict with the grammar of the host language, making it difficult to compose hybrid code except at the level of strings; few mechanisms (if any) exist to control the scope of usage of multiple DSLs; and, most seriously, existing host language tools are typically unaware of the DSL extensions, thus hampering the development process. *Language boxes* address these issues by offering a simple, modular mechanism to encapsulate (i) compositional *changes* to the host language, (ii) transformations to address various *concerns* such as compilation and syntax highlighting, and (iii) *scoping* rules to control visibility of fine-grained language extensions. We describe the design and implementation of language boxes, and show with the help of several examples how modular extensions can be introduced to a host language and environment.

1 Introduction

As domain-specific languages (DSLs) [1] are becoming mainstream, pressure is increasing for better development support and close integration with the host language and existing tools. However today's general-purpose mainstream languages lack the possibility to introduce and express domain-specific concerns in a compact and modular way. Repetitive boilerplate code bloats the code base and makes it difficult to maintain and extend a software system. For example, a single addition of a domain-specific language feature requires changes in both the compiler and the editor. Extending a language to support domain-specific additions thus results in crosscutting changes to the language and development environment. Furthermore different language additions should be active at the same time and tightly integrate with each other, without interfering with each other.

* In Software Language Engineering: Second International Conference, SLE 2009, Denver, Colorado, October 5-6, 2009, LNCS 5969, p. 274–293, Springer, 2009.

Our approach. In this paper we present the model of *language boxes* and how to apply the concepts to embedded languages. Language boxes are used to describe and implement language features in a modular way. Our model works on an executable grammar model [2] of the host language. A *language change* is used to specify a composition of this grammar together with the grammar of a different language. *Language concerns* denote a transformation from parse tokens to the abstract syntax tree (AST) nodes of the host language. Other concerns are supported to specify additional behavior of the tools, such as syntax highlighting, contextual menus, error correction or autocompletion. The *language scope* describes the contexts in which the new language features are enabled. Language boxes yield a high-level model to cleanly embed language extensions and language changes into an existing host environment.

Our contribution. This paper presents the model and implementation of language boxes, a novel model for language engineering and domain-specific language development. Our contributions are the following:

- We use executable grammars to enable fine-grained language changes, language composition and language re-use.
- We define a composable model of language changes and their transformation to the host language.
- We describe the integration of different language related tools, such as editors and debuggers.
- We propose a scoping mechanism to define when and where different language features are to be active.

Outline. This paper is structured as follows: In Section 2 we present a motivating example. In Section 3 we introduce language boxes. Section 4 gives an overview of the implementation identifying the general principles and techniques necessary to build the proposed system. Section 5 shows the implementation in action. Section 6 discusses related work, and Section 7 evaluates and summarizes our approach to introduce new features to an existing language.

2 Language Boxes in Practice

In this section, we demonstrate a simple language extension to motivate our work. As host language we use Smalltalk [3]. Readers unfamiliar with the syntax of Smalltalk might want to read the code examples in the following sections aloud and interpret them as normal sentences. A message send (method invocation) with two arguments is written like this: `receiver do: argument1 with: argument2`; the name of this message (method) is `do:with:`. A message send with no arguments is written like this: `receiver message`. The most important syntactic elements of Smalltalk are the dot to separate statements: `statement1 . statement2`; square brackets to denote code blocks (anonymous functions): `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret: `^` returns the result of the following expression.

The Smalltalk programming language does not include a literal type for regular expressions. Traditionally regular expressions are instantiated by passing a

string to a constructor method of the class `Regexp`. To match a sequence of digits one would, for example, write: `Regexp on: '\d+'`. For developers such lengthy code is repetitive to write. Furthermore, the code is inefficient as the regular expression is parsed and built at run-time. In this section we propose a language extension that adds regular expression literals to the language. This makes a good illustration for our framework, because regular expressions represent an already existing non-trivial domain-specific language that is currently not well integrated into the host system.

A new language box is created by subclassing `LanguageBox`. We use ordinary methods to define the characteristics of the language extension. In our example we start by creating a new language box called `RegexpLanguageBox`. We add the method `change:` returning a change object that determines how to transform the host language grammar.

```
RegexpLanguageBox>>change: aGrammar
   ^ LanguageChange new
      after: aGrammar literal;
      choice: '/' , '/' not star , '/'
```

The first line in bold is the method declaration with class and method name. The returned change specifies that the grammar fragment `'/' , '/' not star , '/'` is appended as an additional choice after the existing grammar production for literals. `aGrammar literal` returns the original production used to parse literal values in the host language.

The grammar extension is defined using a DSL for parser combinators, where the comma is used as a sequence operator and strings denote parsers for themselves. `'/' not star` is a parser that accepts zero or more occurrences of characters other than the slash. In this example the parser accepts any sequence of characters that start and end with the slash delimiter.

The editors, compiler and debugger will automatically pick up the language box and use its change definition to transform the grammar of the host language. Anywhere in the source code where a literal is expected a regular expression with the specified syntax is accepted as well. At this point, the language box does not yet specify any additional behavior for the tools. This means that the compiler would accept code that uses the new language extension, but generate and insert a default null node into the host language AST.

```
RegexpLanguageBox>>compile: aToken
   ^ (Regexp on: aToken string) lift: aToken
```

The above method is a hook method that is automatically called by the compiler to transform the parse tree tokens of the language extension to the host language AST. In our example we instantiate a regular expression object from the token value. The method `lift:` takes the regular expression object and wraps it into a literal node of the host language. The original token is passed into the literal node to retain the source mapping for the debugger.

Expressions like `'One After 909' =~ /\d+/` can now be used and evaluated from anywhere within the system. As depicted in Figure 1 the transformed grammar of the host language parses the source code and uses our custom transformation function `compile:` as a new production action to transform the input to

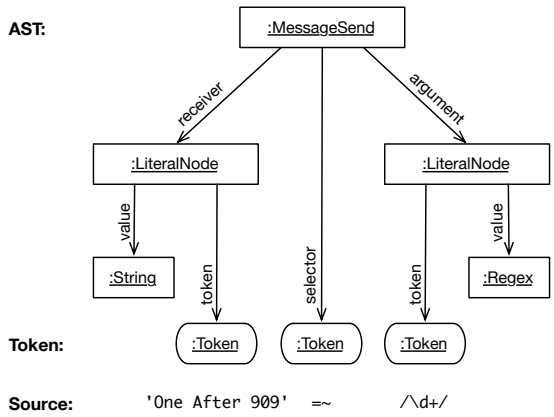


Fig. 1: From the source code to the AST of the host language.

the host AST. Note that `=~` is a matching operator with respect to regular expressions. This operator is not a language extension, but a method implemented in the `String` class. In this example, the matched sub-string `'909'` is returned.

The syntax highlighter in the editor recognizes the regular expression syntax as valid, but it still colors the source using the default font style. To change that, we add syntax highlighting instructions to the language box:

```
RegexpLanguageBox>>highlight: aToken
   ^ aToken -> Color orange
```

With only a few lines of code we have demonstrated how to extend the syntax of a general purpose language with a new literal type, how to define the transformation to the host language AST and how to integrate it into editors by customizing the syntax highlighting.

3 Language Box Model

Parser, compiler and associated development tools are usually black boxes. Extending, changing or replacing the default behavior is not easy and thus discouraged. We propose a high-level language model that provides us with fine grained access to the different syntactic elements of the host language without revealing too much of the underlying implementation. Furthermore we provide a set of extension points for the language grammar to allow developers to extend the compiler and available development tools. Language extensions should be open, in the sense that they tightly integrate anywhere in the host language grammar without requiring special syntactical constructs to change between different language extensions.

As depicted in Figure 2 the language box model consists of three parts: In Section 3.1 we introduce the *language change*, which defines how the grammar of the host language is changed. Then in Section 3.2 we explain how *language concerns* customize the behavior of language extensions, such as syntax highlighting

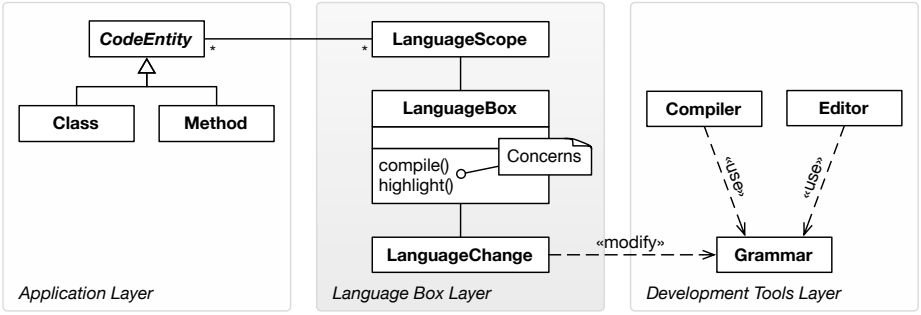


Fig. 2: The interplay of the language box model with the application layer and the development tools.

or menu actions. Finally in Section 3.3 we discuss the *language scope*, which is used to restrict the effect of a language box to certain parts of the application code.

3.1 Language Change

The language change is used to encapsulate the change applied to the grammar of the host language. In our case the language extension is defined using a grammar fragment and a specification of how this fragment is composed with the grammar of the host language.

In Section 2 we added a new regular expression literal as an additional choice to the existing literals. This means the host language grammar rule was changed from

```
Literal ::= String / Number / Boolean
```

to

```
Literal ::= String / Number / Boolean / Regexp
```

where `Regexp` was defined as `Regexp ::= '/' , '/' not star , '/'`. In addition to appending to the end of a choice, we also support various other composition strategies to combine the grammar of the host language and the new grammar fragment. These composition strategies are listed in Table 1.

An important property of the language change is that the grammar that is composed into the host language might reference other productions from the existing grammar. This allows language designers to reuse existing features of the host language and closely integrate existing syntax with the language extension. Depending on the host language production we can decide to change the language box to replace the complete host language with a new grammar (for example when the start production of the grammar is replaced), or just to change subtle features (for example when adding a new literal type).

While the inserted grammar fragment in our initial example was intentionally chosen to be trivial, it is possible to compose arbitrary complex grammars using the given composition strategies. Furthermore multiple composition strategies can be defined in the same language box, as we will demonstrate in Section 5 where Smalltalk and SQL are combined.

Action	Composition	Production
replace	–	$R ::= X$
before	sequence	$R ::= X\ A$
after	sequence	$R ::= A\ X$
before	choice	$R ::= X\ /\ A$
after	choice	$R ::= A\ /\ X$

Table 1: Composition strategies for a grammar rule $R ::= A$. A is a symbol of the original grammar. X is the extending grammar fragment as defined by the language change. $X\ A$ denotes the sequence of X and A, $X\ /\ A$ denotes an ordered choice between X and A.

3.2 Language Concern

When changing or adding new language features, there are different concerns to integrate into the toolset of the application developers. First and foremost we need to specify a transformation from our language extension to the code representation of the host language. Optionally we might want to closely integrate the language extensions into the existing programming tools, such as editors and debuggers. This is done by adding concerns to the language box such as:

- **Compilation.** This concern describes the transformation from the AST nodes and parse tree tokens of the language extension to the AST of the host language. We call this process *compilation* because it makes the language extension executable. Subsequently the host language AST is passed into the standard compiler tool-chain that compiles it further down into an efficiently executable representation.
- **Highlighting.** The syntax highlighter concern annotates the source ranges with color and font information, so that the editor and debugger are able to display properly colored and formatted text. The resulting source ranges and styling information is then passed into the standard editors for display.
- **Actions.** This concern provides a list of labels and associated actions that are integrated into the standard contextual menu of editors. This allows for context sensitive functionality, such as language specific refactorings. Thus unsuitable actions from the host language or other language extensions are not displayed when the user works in the context of a language extension.

Other concerns can be specified similarly, for example enhanced navigation and search facilities, error correction, code expansion templates, code completion, code folding, or pretty printing.

Concerns are implemented by overriding a default implementation. This facilitates the evolution of new language features, starting from a minimal language box that defines a change to the host grammar only. At a later point the language designer can incrementally add new concerns to make the language integrate more appropriately with the tools. In the introductory example we saw

that the compilation and highlighting concerns were not specified in the beginning. In this case a default implementation caused the compiler to insert a null node and the highlighter to use the default text color.

3.3 Language Scope

To scope the effect of language boxes to well-defined fragments of the application source code, we need a way to specify the extent of the language changes within the application code. The scope identifies language boxes and the associated code entities, as depicted in Figure 2. Language developers can define a default scope. From coarse to fine grained the following scopes are supported:

- **System.** The system scope affects all the source code of the system without restriction. This is the default, if no more restrictive scope is specified.
- **Package.** The package scope affects all source artifacts contained in a particular package.
- **Class.** The class scope affects all source artifacts of a particular class, or its class hierarchy.
- **Method.** The method scope affects a particular method, or methods with a particular name.

Furthermore we give the language box users the possibility to explicitly add a language box to a particular code entity (package, class, method) or to remove it. This effectively overrides the default scope and facilitates a fine-grained control of language features from within the application code. Language boxes are added or removed using either a context menu in the user interface or a declarative specification in the source code.

Whenever a tool requests a grammar at a specific location in the source code, the language box system determines all active language boxes by comparing their scope with the current source location. It then transforms the host language grammar according to the change specification in the language boxes and inserts the concerns for the active tool. This enables one to scope language boxes and their grammar changes to well-defined parts of the system.

4 Implementation

To validate the language box model, we have implemented it in Pharo [4], an open-source Smalltalk [3] platform[3]. Language boxes are implemented on top of the extensible compiler framework Helvetia. As host language we use Smalltalk because it provides excellent access to compiler and tools, as everything is implemented in Smalltalk itself and is accessible at runtime [5]. Furthermore the syntax of Smalltalk is relatively simple (*i.e.,* 11 AST nodes, 52 grammar rules), which makes it a good base for program and grammar transformation.

Our implementation depends on the following host language features:

[3] Our implementation of language boxes along with its source code and examples can be downloaded from `http://scg.unibe.ch/research/helvetia`.

Modular compiler. The internals of the compiler must be accessible so that a custom parser and an additional transformation phase can be introduced.

Structural reflection. The system must provide the capability to query packages, classes, and methods to determine when and where to apply language boxes.

Behavioral reflection. The AST must be a first class abstraction that can be queried, extended with new node types, built and transformed during compilation.

Extensible tools. The development environment and its tools must be extensible and have full access to structural and behavioral reflection.

Our implementation of language boxes is lightweight because we reused as much functionality from the host environment as possible. Our approach is entirely implemented in Smalltalk. We do not change the underlying virtual machine. The implementation presented in this paper consists of 640 lines of Smalltalk code, of which 410 lines consist of a reimplementation of the traditional Smalltalk parser as an executable grammar. Please refer to our related work for a detailed comparison of different programming languages and their support for our requirements [6].

To facilitate transformations on the host language grammar we replaced the standard LALR parser of Smalltalk with our own implementation. We combine four different parser methodologies: scannerless parsers, parser combinators, parsing expression grammars and packrat parsers.

- *Scannerless Parsers* [7] combine lexical and context-free syntax into one grammar. This avoids the common problem of overlapping token sets when grammars are composed. Furthermore language definitions become more concise as there is only one uniform formalism.
- *Parser Combinators* [8] are building blocks for parsers modeled as a graph of composable objects; they are modular and maintainable, and can be changed, recomposed and reflected upon. Parser combinators enable language boxes to perform grammar transformation on the fly.
- *Parsing Expression Grammars* (PEGs) [9] provide ordered choice. Unlike in parser combinators, the ordered choice of PEGs always follows the first matching alternative and ignores other alternatives. With valid input we always get exactly one parse-tree, the result of a parse is never ambiguous.
- *Packrat Parsers* [10] give us linear parse time guarantees and avoid problems with left-recursion in PEGs. We discovered that enabling memoization on all primitive parsers is not worth the effort, as the additional bookkeeping outweighs the performance gain. We only apply memoization at the level of productions in the language definition, which significantly improves parsing speed.

Before a method is compiled, a custom parser for that particular compilation context is composed. This new parser is built by starting from the standard grammar of the host language and by applying the change objects of the active language boxes in the defined order.

Figure 3 depicts a fragment of the original Smalltalk grammar and the regular expression language extension that we introduced in Section 2. The composition

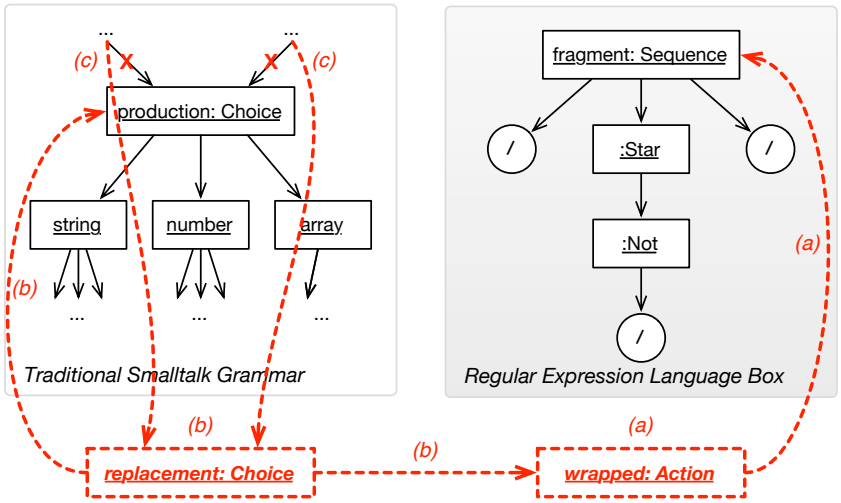


Fig. 3: Traditional Smalltalk (left) and the regular expression extension (right) are combined in three steps to a single grammar: (a) the grammar fragment is wrapped with the action, (b) the wrapped fragment is combined with the existing grammar, and (c) all references to the original production are replaced with the combined one.

algorithm takes the original grammar of the host language `aGrammar` and the grammar of the language extension `fragment` and composes them using the following algorithm. For conciseness we present the complete algorithm as a single method with nested conditional statements instead of the original implementation, which makes use of the strategy design pattern.

```
LanguageChange>>modify: aGrammar with: aLanguageBox
   | wrapped replacement |
   wrapped := fragment ==> [ :nodes |                    " Figure 3(a) "
      aLanguageBox
         perform: aGrammar concern
         with: (self transform: nodes) ].
   replacement := action = 'replace'                     " Figure 3(b) "
      ifTrue: [ wrapped ]
      ifFalse: [
         action = 'before'
            ifTrue: [ composition with: wrapped with: production ]
            ifFalse: [
               action = 'after'
                  ifTrue: [ composition with: production with: wrapped ]
                  ifFalse: [ self error: 'Invalid composition.' ] ] ].
   aGrammar replace: production with: replacement        " Figure 3(c) "
```

The composition algorithm `modify:with:` is implemented in the language change object. As input parameter the method takes the original language gram-

mar `aGrammar` and the language box `aLanguageBox` responsible for this change. The actual transformation is a three step process:

1. Lines 3–6 fetch the grammar fragment and wrap it with the concern of the language box. This is achieved with the `==>` operator which adds an action to a production. In our example with the regular expressions the fragment is the new parser `'/' , '/' not star , '/'`. The concern depends on what the grammar is used for. If the grammar is used for compilation, the compile concern `compile:` is called; if the grammar is used for syntax highlighting, the highlight concern `highlight:` is called, *etc.* This does not change the structure of the resulting language grammar, but allows the production actions to produce different results for the different concerns. While the compilation concern requires a complete and valid AST the highlighting concern produces a stream of tokens with source position and color information.
2. Depending on the `action` and the selected `composition` a new grammar fragment is built (Figure 3(b)):
   (a) The *replace action* (line 8) replaces the selected grammar production with the wrapped fragment.
   (b) The *before action* (line 11) composes the wrapped fragment with the old production using either choice or sequence as composition operator.
   (c) The *after action* (line 14) composes the old production with the wrapped fragment using either choice or sequence as composition operator.

   In our example the replacement production is defined as a choice that is added after the original literal production, *i.e.,* `replacement ::= production / wrapped` where `production` is the grammar fragment for literals in the original Smalltalk grammar.
3. Last on line 16 (Figure 3(c)) the grammar is told to replace all references to the original production with the replacement. This is done by traversing the complete grammar and replacing all the references to the old production with the new one. In our example all references to original literal production are replaced with the newly composed grammar fragment. This step ensures that the new grammar can parse regular expressions everywhere the host syntax would expect a literal.

In Smalltalk, the unit of editing and compilation is the method. This facilitates our language box model and enables a straightforward integration with the editor and other tools. The small and well-defined unit of editing eases the way for language boxes, however it is not a strict requirement. Depending on the granularity of the language scopes to be supported, grammar changes could be applied at the level of packages, files, classes or methods. The scoping of language boxes depends on the reflective capabilities of the host system [11].

5 Case Study

In order to demonstrate the applicability of language boxes in practice, we present and discuss a more elaborate language extension. The goal is to embed a subset of the *Structured Query Language* (SQL) in the host language.

Furthermore SQL should be extended so that values within the query can be safely replaced with expressions from the host language.

The following method shows a typical example of an embedded SQL query:

```
SQLQueries>>findUser: aString
   | query rows |
   query := 'SELECT * FROM users WHERE username = "' , aString
    asEscapedSql , '"'.
   rows := SQLSession execute: query.
   ^ rows first
```

The query is concatenated from a series of strings and the input parameter `aString`. The composition of SQL from strings is not only error prone and cumbersome, but also introduces possible security exploits. The developer has to pay attention to correctly escape all input, otherwise an attacker might be able to manipulate the original SQL statement.

The following method shows the improved version using a language box for SQL statements:

```
SQLQueries>>findUser: aString
   | rows |
   rows := SELECT * FROM users
           WHERE username = @(aString).
   ^ rows first
```

SQL statements can be used anywhere in the host language where an expression is expected. The syntax of the SQL expression is automatically verified when compiling the code, assembled and executed, and the result is passed back into the host language as a collection of row objects. SQL itself is extended with a special construct `@(...)` to embed host language values into the query.

5.1 Adding an SQL Language Extension

Since SQL is a language on its own and considerably more complex than the regular expression language we saw before, we use an external class to define its grammar. To do this we took advantage of the same infrastructure that we used to define a mutable model of the host language grammar. We implemented the syntax specification described for SQLite[4] which is almost identical to SQL92, but leaves out some of the more obscure features.

To combine the host language and SQL, we create a new language box called `SQLanguageBox`. Again we specify a change method that describes how the new language is integrated into the host language:

```
1 SQLanguageBox>>change: aSmalltalkGrammar
2    | sqliteGrammar compositeChange |
3    sqliteGrammar := SQLiteGrammar new.
4    compositeChange :=
5         (LanguageChange new
6             before: aSmalltalkGrammar expression;
```

[4] `http://www.sqlite.org/syntaxdiagrams.html`

```
7               choice: sqliteGrammar)
8       + (LanguageChange new
9               before: sqliteGrammar literalValue;
10              choice: '@(' , aSmalltalkGrammar expression , ')').
11  ^ compositeChange
```

On line 3 we instantiate the SQL grammar defined in the class `SQLiteGrammar`. In this example a single grammar transformation is not enough. On lines 5–7 we extend the production for host language expressions with SQL as an additional choice that is added before the original expression production. On lines 8–10 we extend the production for SQL literal values with a new syntax that lets Smalltalk expressions be part of SQL. The two changes are composed using the `+` operator and returned on line 11.

Note that the first change object introduces ambiguity into the host language grammar. Intentionally we decide that the SQL grammar should take precedence over the Smalltalk expression production and insert the SQL grammar before the expression production of the host language. `SELECT * FROM users` is both a syntactically valid SQL statement and a syntactically valid Smalltalk expression[5]. Since we added the SQL grammar to the beginning of the original host language production any expression is first tried with the SQL grammar. If that does not work the original production of the host language expression will take over. The ordered choice of PEGs avoids the ambiguity by giving SQL precedence over Smalltalk.

The problem that an SQL expression can potentially hide valid Smalltalk code remains open. The current implementation gives the responsibility to detect and avoid such problems to the language developer. Language boxes provide the tools to tightly control the scope of language changes, as discussed in Section 3.3. Furthermore, conflicting language changes can always be surrounded by special tokens to make the intention absolutely clear. An example of this can be seen in the example above on line 10 where Smalltalk expressions in SQL are surrounded by `@(...)`. If possible we try to avoid such extra tokens as they clutter the close integration of the new language. When integrating SQL into Smalltalk this is less of a problem, as SQL is a very strict language with a rigid and very verbose syntax. A test run on a large collection of open-source Smalltalk code with a total of over 1 200 000 expression statements revealed that none of them parsed as valid SQL.

Similar to the regular expression example we define a compilation concern that tells the language box how to compile the new expression to the host language. In this example we do not receive a single token, but the complete AST as it is produced by the SQL grammar.

[5] In Smalltalk, this would send the message `users` to the variable `FROM`, and then multiply the result with the variable `SELECT`.

```
1 SQLanguageBox>>compile: anSQLNode
2    | nodes query |
3    nodes := anSQLNode allLeaves collect: [ :token |
4       token isToken
5          ifTrue: [ token string lift: token ]
6          ifFalse: [ ``(`,token asEscapedSql) ] ].
7    query := nodes fold: [ :a :b | ``(`,a , ' ' , `,b) ].
8    ^ ``(SQLSession execute: `,query)
```

The method above makes use of quasiquoting facilities known from Lisp [12] and OMetaCaml's staging constructs [13]. A quasiquote is an expression prefixed with `` which is delayed in execution and represents the AST of the enclosed expression at runtime. The unquote `, is used within a quasiquoted expression. It is executed when the AST is built and can be used to combine smaller quasiquoted values to larger ones. This language extension has also been realized using language boxes.

The compilation concern flattens all the leaf nodes of the SQL AST (line 3) and transforms the input to host language AST nodes (lines 4–6). Tokens of the SQL AST are transformed to literal nodes in the host language (line 5). If the node comes from embedded Smalltalk code, we automatically wrap the expression with a call to `asEscapedSql` to ensure it is properly escaped (line 6). Finally we concatenate all the nodes to a single query expression (line 7), which is then sent to the current SQL session manager (line 8).

5.2 Restricting the Scope of a Language Extension

As we noted in our introductory example, by default a language box is active in the complete system. In many cases this is not desired, especially when a language change is more intrusive. We provide two different ways of modeling the scope of a language extension. While the first one is aimed at language designers, the second one targets language users who want to select and activate available language extensions while working in their code.

The *language designer* can specify a scope for a language, by overriding the `scope` method in the language box. The default implementation of the method returns a system scope, but frameworks might want to reduce the scope to certain class hierarchies, packages, classes or methods. This feature makes use of the reflective facilities of the host language to determine if a given language box is active in a specific compilation context.

The *language user* can further change the scope of a language box through the code editor. As an extension we added menu commands that allow developers to add and remove language extensions from code entities like packages, classes and methods. This is useful for language extensions that make sense in different smaller scopes that cannot be foreseen by the language designer. Furthermore we extended the code editor with functionality to display the active extensions, so that the developer knows what he is expected to write. Also distinct syntax highlighting (*i.e.,* different background colors) in the language definition can help developers to know in which context they are currently working.

5.3 Mixing Different Language Extensions

The SQL language extension blends nicely with the host language, as well as the regular expression language extension we presented previously. For example we can use both language extensions at the same time, together with the host language:

```
rows := SELECT * FROM users
        WHERE username = @(aString ~= /\s*(\w+)\s*/)
```

This example transparently expands to the following (more verbose) code:

```
rows := SQLSession execute: 'SELECT * FROM users WHERE username = ' ,
   (aString ~= (Regexp on: '\s*(\w+)\s*')) asEscapedSql
```

The compiler automatically ensures that the SQL statement is syntactically valid, that all values injected into the statement are properly escaped and that the query is automatically executed within the current session.

5.4 Tool Integration

Adding syntax highlighting to the SQL expressions is straightforward. Contrary to the regular expression that was highlighted using a single color, the SQL extension is more complex and we have to deal with many different kinds of keywords, operators and expression types. To avoid having to specify the highlighting for every production within the language box itself, we allow language developers to specify an external class that specifies the concern-specific production actions. In the case of syntax highlighting these actions return the color and font information.

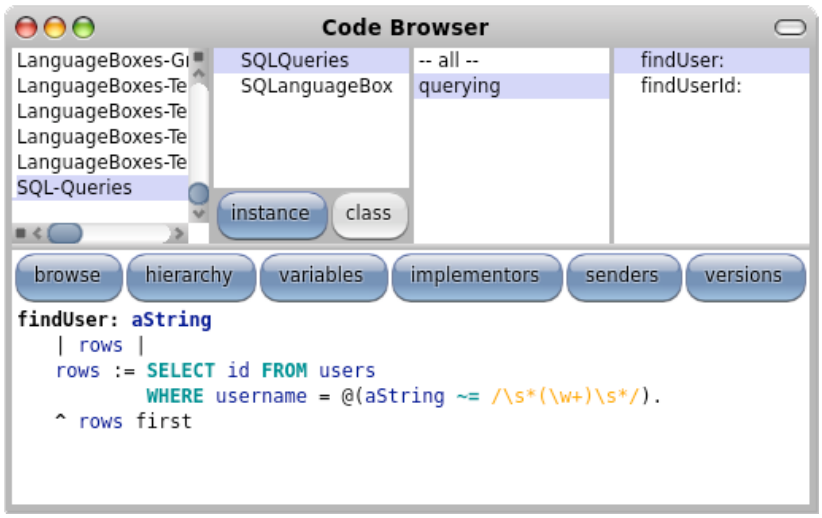
```
SQLanguageBox>>highlight
   ^ SQLiteHighlighter
```

Adding a context menu item that links to the SQL documentation is a matter of adding the method:

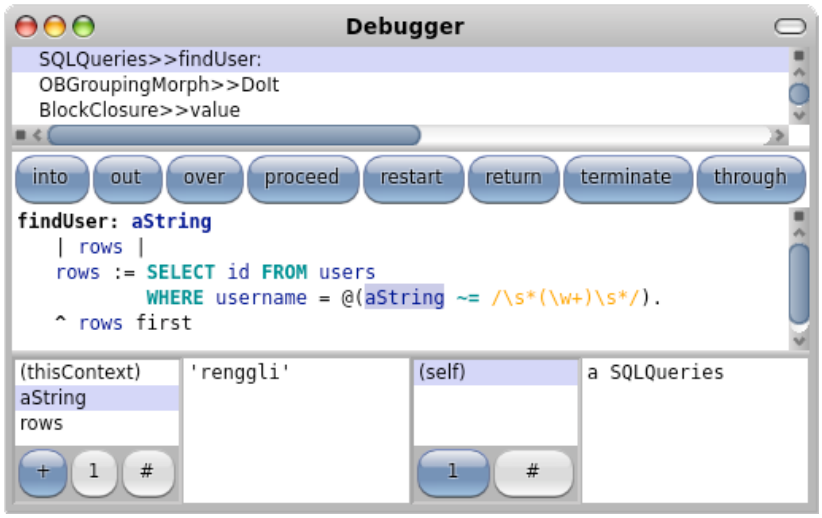
```
SQLanguageBox>>menu: aMenu using: anSQLNode
   ^ aMenu
      addMenuItem: 'SQLite Documentation'
      action: [ WebBrowser open: anSQLNode documentationUrl ]
```

Clicking on the menu item opens a web browser on the URL returned by the AST node under the cursor. The method `documentationUrl` is implemented to dispatch to the parent node if no documentation is available at the most specific AST node.

Figure 4 depicts a standard Smalltalk code browser and a debugger on the presented example. The upper part of both windows show the navigation context, in the code browser this is the currently edited package, class and method; in the debugger this is the execution stack. In both cases the lower part shows the source code of the method properly highlighted.



(a) Code Browser



(b) Debugger

Fig. 4: Development tools on a method that combines two language boxes and the host language.

The transformations as defined by the compilation concern are not visible to end-users that work at the level of source-code. The transformations are only visible at the level of the compiled bytecode code. All tools, including the debugger, display the original source code only. Stepping through custom languages in the debugger works similarly to traditional Smalltalk. Since all our transformations are on the level of the standard AST nodes and tokens, their original location in the source code can be traced back. The use of the AST to highlight the current execution position is a standard feature of the debugger. Generated nodes that do not have a physical position in the source code are ignored when stepping through with the debugger.

Another example of how tightly language boxes integrate into the host language are breakpoints. In traditional Smalltalk breakpoints are implemented by invoking the method `halt` at the desired position in the source code. This method is implemented by the system. It stops the execution of the active process and opens a debugger at the current execution point. Since breakpoints are implemented at the level of AST nodes, they continue to work even within language extensions. Upon execution of a `halt` instruction the debugger opens up and automatically highlights the currently active code statement.

6 Related Work

There is a wide range of academic and commercial products that facilitate the creation and integration of DSLs. We categorize some of these systems.

Meta-programming Systems. *Converge* [14], the *Extensible Programming Language* (XMF) [15], and *Katahdin* [16] encourage the integration of new languages using their meta-programming facilities. All these languages provide their own proprietary host languages. Converge and XMF require special syntactic tokens to change to a DSL, and therefore do not allow developers to arbitrarily extend and mix host language and DSLs. With Katahdin it is possible to override and replace grammar rules of the host language. None of these tools provides integration into a programming environment.

ASF+SDF [17] is a language neutral collection of tools for the interactive construction of language definitions and associated tools. SDF is implemented as a scannerless generalized LR parser and supports the composition of grammars. A single parse table is created for all possibly active productions, and depending on the context the corresponding transitions are enabled and disabled. Our use of parser combinators allows us to directly model the grammar as an executable graph of productions that can be recombined and modified on the fly.

Language Workbenches. Language workbenches [18] provide sophisticated environments to design new languages and to use them with existing tools. The *Meta Programming System* (MPS) provides a convenient environment to define new languages. Languages are specified using multiple interleaving concepts to define grammar, editors, constraints, transformations, behavior, type systems, data flow and code generators. Language workbenches are typically generative

frameworks, which means that the code is transformed down to source files of the host language before compilation and execution. This is in strong contrast to our approach where we transform a common AST representation with exact source information.

Xtext is a framework for development of textual domain-specific languages. It is integrated well with the Eclipse environment and especially the Eclipse modeling tools, but it does not provide the possibility to change the Java programming language itself.

Generative Approaches. *TXL* [19] is a source transformation system for experimentation with language design. A TXL program consists of a base grammar definition and a series of overrides that extend and change the base grammar. These language changes are global, while various traversal and rewrite strategies can be contextually scoped to perform a source-to-source transformation. Our model provides a high-level concept of language changes that are augmented with different transformation concerns for compiler and tool integration. Our target is always the host language AST, that is directly used to generate executable code.

MetaBorg [20] is a method for embedding DSLs and extending existing languages. MetaBorg is based on the *Stratego/XT* [21] toolkit, a language independent program transformation engine. MetaBorg employs a scannerless generalized LR parser technique to compose different grammars, and an annotated term language to build abstract syntax trees. While this approach is language independent, it is also much more complex than our implementation. Our use of a parser combinator library makes it straightforward to define and transform arbitrary context-free grammars, ambiguities are supported and automatically resolved by the rule order. To define the transformation, MetaBorg uses a quoting mechanism similar to ours, however the resulting code is pretty printed to a string before passing it on to the compiler of the host language. Hence there is no close integration in the compiler, the development environment or code debuggers.

MontiCore [22] uses language inheritance and language embedding [23] to facilitate modularity. Language inheritance enables changing and adding grammar rules to a super grammar but has the problem that multiple extensions cannot be easily combined without resorting to multiple inheritance or delegation. Language embedding enables the composition of existing grammars. MontiCore does not model the host language (Java) as a first class entity, so extending the host language is not directly possible.

The *Linglet Transformation System* [24] provides a mechanism to modularize the syntax and semantics of a single language construct. The code to generate is specified using a templating system. Linglets can be composed with each other and integrated into the host language at specific extension points. There is no support to replace or change existing language features and no scoping mechanism. Contrary to our approach the linglets are only used during compilation; other tools do not take advantage of the language model.

Attribute Grammars. Van Wyk *et al.* [25] propose forwarding attribute grammars to catalyze modularity of language extensions. The *Java Language Extender* framework [26] is the tool that uses this technique to import domain-adapted languages into Java. The use of LR-style parsers enforces certain restrictions to imported extensions, as the resulting grammar needs to be unambiguous. Language extensions can be scoped to files, but not at a more fine-grained scale. Language boxes uses a form of attribute grammars too, where new constructs are expressed as semantically equivalent constructs of the host language. The Java Language Extender framework does not provide an integration into the IDE.

7 Conclusion

In this paper we have presented language boxes, a novel model to bend the syntax and semantics of the host language. We have presented the concepts, an implementation and two examples of language boxes. We have demonstrated how language boxes encapsulate language extensions and enable mixing different language changes. We have further demonstrated how existing tools are closely integrated with new language features.

The solution proposed in this paper has the following properties:

Model. The language box model encapsulates changes to the grammar of the host language and defines different concerns that specify the behavior of the language extension in the tools. The scope defines the context in the source code where the language extension is active.

Modular. Language boxes are modular. Language extensions can be independently developed and deployed. The use of parser combinators makes it possible to combine grammars and even to support ambiguous ones in a meaningful way.

Concerns. Tools can be extended with language specific concerns. Language extensions can be developed incrementally. While the compilation concern is usually defined first, editor integration can be provided later.

Homogeneous Language Integration. Language boxes use the abstract code representation of the host language, different languages can be arbitrarily composed, access the same data and pass control to each other.

Homogeneous Tool Integration. The IDE, and especially the debugging tools, continue to work and actively support the language extensions. Stepping through a mixture of code from different languages poses no problem either. Changing and recompiling the source code on the fly from within the debugger is viable, this being an inherited feature from the host language.

Language boxes provide a model to extend the host language and as such are well suited to define embedded DSLs [27]. Language boxes are implemented in the host language, and are thus an internal domain-specific language themselves. This makes our approach adaptable to new requirements, as well as enabling a close integration with the host language.

The language box compiler is twice as slow as the traditional compiler, because a custom parser has to be composed for every compilation context. The

parsing itself is not noticeably slower than with the LALR parser and there is no visible lag even for syntax highlighting, as methods tend to be short and memoizing packrat parsers guarantee linear time. To improve the speed of batch compilation, *i.e.,* when loading an external package, we plan to add grammar caches in a further release.

Different language boxes can potentially influence or conflict with each other and the host language. We could rarely observe this problem in practice though, since most language changes are clearly scoped and often affect different parts of the original grammar. Two language boxes that add new literal types could result in a potentially ambiguous grammar where one language extension hides another one. In this case the language extension that was loaded last will take precedence over the language extension loaded earlier. This could introduce unexpected side-effect into the code of the user. As future work we plan to investigate into ways of detecting and notifying the user about such problems upfront.

Language boxes are not yet used in industrial projects, but we have successfully applied them to a variety of problems in our own domain. For example the definition of the grammars and the quasiquoting functionality in the language boxes implementation itself is implemented using language boxes. As future work we plan to validate our approach on a wide variety of other language extensions that we have collected from an industrial context. We also want to look into ways to automatically refactor code from the host language towards a DSL.

Furthermore, we plan to apply the language box model on other programming languages. We currently have a prototypical pre-compiler for Java that can be used to parse a file with a transformed grammar and to pretty print the result to standard Java code. While this does not provide the necessary tool integration and fine-grained scoping rules yet, it demonstrates that our language box model is viable for statically typed languages with a considerably more complex syntax.

Acknowledgments

We thank Alexandre Bergel, Tudor Gîrba, Adrian Lienhard, and Jorge Ressia for their 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).

References

1. Fowler, M.: Domain specific languages (June 2008) http://martinfowler.com/dslwip/, Work in progress.
2. Bracha, G.: Executable grammars in Newspeak. Electron. Notes Theor. Comput. Sci. **193** (2007) 3–18
3. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (May 1983)
4. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates (2009)
5. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. In: Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007. Volume 6/9., ETH (October 2007) 231–251

6. Renggli, L., Gîrba, T.: Why Smalltalk wins the host languages shootout. In: Proceedings of International Workshop on Smalltalk Technologies (IWST 2009), ACM Digital Library (2009) To appear.
7. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)
8. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
9. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2004) 111–122
10. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. Volume 37/9., New York, NY, USA, ACM (2002) 36–47
11. Foote, B., Johnson, R.E.: Reflective facilities in Smalltalk-80. In: Proceedings OOPSLA '89, ACM SIGPLAN Notices. Volume 24. (October 1989) 327–336
12. Bawden, A.: Quasiquotation in Lisp. In: Partial Evaluation and Semantic-Based Program Manipulation. (1999) 4–12
13. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. (2003) 30–50
14. Tratt, L.: The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London (February 2005)
15. Clark, T., Sammut, P., Willans, J.: Superlanguages, Developing Languages and Applications with XMF. Volume First Edition. Ceteva (2008)
16. Seaton, C.: A programming language where the syntax and semantics are mutable at runtime. Technical Report CSTR-07-005, University of Bristol (June 2007)
17. Klint, P.: A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology (TOSEM) **2**(2) (1993) 176–201
18. Fowler, M.: Language workbenches: The killer-app for domain-specific languages (June 2005) `http://www.martinfowler.com/articles/languageWorkbench.html`.
19. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3) (2006) 190–210
20. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D.C., ed.: Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, Canada, ACM Press (oct 2004) 365–383
21. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C., et al., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. Spinger-Verlag (June 2004) 216–238
22. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular development of textual domain specific languages. In Paige, R., Meyer, B., eds.: Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Springer-Verlag (2008) 297–315
23. Spinellis, D.: Notable design patterns for domain specific languages. Journal of Systems and Software **56**(1) (February 2001) 91–99
24. Cleenewerck, T.: Component-based DSL development. In: Proceedings of the 2nd international conference on Generative programming and component engineering, Springer-Verlag New York, Inc. New York, NY, USA (2003) 245–264
25. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. Lecture Notes in Computer Science (2002) 128–142

26. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. Lecture Notes in Computer Science **4609** (2007) 575
27. Hudak, P.: Building domain specific embedded languages. ACM Computing Surveys **28**(4es) (December 1996)