MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT

Martin Bravenboer, René de Groot, and Eelco Visser

Department of Information and Computing Sciences,
Universiteit Utrecht, P.O. Box 80089 3508 TB, Utrecht, The Netherlands
{martin, rcgroot, visser}@cs.uu.nl

Abstract. General-purpose programming languages provide limited facilities for expressing domain-specific concepts in a natural manner. All domain concepts need to be captured using the same generic syntactic and semantic constructs. Generative programming methods and program transformation techniques can be used to overcome this lack of abstraction in general-purpose languages.

In this tutorial we describe the METABORG method for embedding domain-specific languages, tailored syntactically and semantically to the application domain at hand, in a general-purpose language. METABORG is based on Stratego/XT, a language and toolset for the implementation of program transformation systems, which is used for the definition of syntactic embeddings and assimilation of the embedded constructs into the surrounding code.

We illustrate METABORG with three examples. JavaSwul is a custom designed language for implementing graphical user-interfaces, which provides high-level abstractions for component composition and event-handling. JavaRegex is a new embedding of regular expression matching and string rewriting. JavaJava is an embedding of Java in Java for generating Java code. For these cases we show how Java programs in these domains become dramatically more readable, and we give an impression of the implementation of the language embeddings.

1 Introduction

Class libraries are reusable implementations of tasks in a certain domain. The library is used via some API, which constitutes a 'language' for using the library implementation. The syntax of this language provided by the API is based on the syntax of the general-purpose language in which the API is used. Unfortunately, general-purpose programming languages provide limited facilities for expressing domain-specific concepts in a natural manner. This syntax of the general-purpose language does not always allow the appropriate notation and composition of domain concepts.

Examples of this issue are available everywhere. For example, user-interface code is typically a tangled list of statements that constructs a hierarchical structure. XML document construction is verbose or unsafe. Java libraries often return `this` only for making sequential composition of calls possible, thereby confusing users, compilers, and other meta-programs. Regular expressions need to be escaped heavily since they have to be encoded in strings. Not to mention the run-time errors or security risks involved in composing SQL, XPath or XQuery queries by concatenating strings [7, 9]. Clearly, this is a serious issue.

Generative programming methods and program transformation techniques can be used to overcome this lack of abstraction in general-purpose languages. To this end, we proposed the METABORG[1] [6] method, which is a general way of providing domain-specific notation for domain abstractions to application programmers. METABORG is a way of implementing an embedding of a domain-specific language in a general-purpose language. METABORG starts off with the idea that there should be no restrictions (1) on the syntactic extension, (2) on the interaction with the host language, and (3) on the translation to the general-purpose code (a process we call assimilation).

In [6] several METABORG examples have been presented, but the implementation of these examples could not be discussed in detail. In this paper, we give a more extensive account of the implementation of three METABORG examples, thus providing more insight in the METABORG method for embedding domain-specific languages. We focus on the METABORG examples and experience we gained from this. For an extensive account of alternative approaches and related work, we refer to [6].

METABORG is based on modular syntax definition in SDF, which is implemented by scannerless generalized-LR parsing [4, 11] and source to source transformation in the high-level language for program transformation Stratego [13]. Stratego is a general-purpose language for the implementation of program transformation systems. On top of a small core language for pattern matching, abstract syntax tree construction, and term traversal, Stratego provides abstractions such as rewrite rules whose application can be controlled by a rewrite strategy. Context-sensitive rewritings are handled by defining rewrite rules dynamically at the location where the context information is available. Stratego is distributed as part of Stratego/XT, which is the combination of the Stratego program transformation language and an extensive set of transformation tools for parsing, pretty-printing, and so on.

In this paper, we present three examples of METABORG applications. These examples illustrate the capabilities of the METABORG method and provide an introduction to the implementation of such embeddings. In Section 2 we given an extensive overview of the implementation of JavaSwul, the embedding of a custom designed language for implementing graphical user-interfaces. In Section 3 we give a short overview of JavaRegex, which is an embedding of regular expression matching and string rewriting, and JavaJava, which is an embedding of Java in Java, intended for Java code generation.

2 Embedding Swul in Java

Swul is a domain-specific language for writing user-interfaces based on the Swing library. In this section, we will discuss in more detail why and how we implemented an embedding of Swul in Java. The Swul implementation described in this section is a major extension of the first sketch of Swul presented in [6].

[1] METABORG provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to strengthen itself. The ease of implementing embeddings makes resistance futile.

```
JMenuBar menubar = new JMenuBar();
JMenu filemenu = new JMenu("File");
JMenuItem newfile = new JMenuItem("New");
JMenuItem savefile = new JMenuItem("Save");
newfile.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N, 2));
savefile.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S, 2));
filemenu.add(newfile);
filemenu.add(savefile);
menubar.add(filemenu);

JPanel buttons = new JPanel(new GridLayout(1, 2));
JPanel south = new JPanel(new BorderLayout());
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);

JPanel panel = new JPanel(new BorderLayout());
panel.add(BorderLayout.CENTER, new JScrollPane(new JTextArea(20, 40)));
panel.add(BorderLayout.SOUTH, south);
```

Fig. 1. Simple user-interface implemented in plain Java

First of all, why did we develop a domain-specific language for implementing user-interfaces? Despite all the advances in user-interface libraries, typical user-interface code is still difficult to read. The composition of a complete user-interface from its basic components is a tangled list of statements, that makes it difficult to see how the user-interface is structured. A typical implementation of a simple graphical user-interface is shown in Figure 1. The composition of the user-interface components and panels in separate statements results in spaghetti-like code: the connections between the definitions and uses of components are unclear. In plain Java, the *implementation* of a graphical user-interface is not close enough to the *domain* of graphical user-interfaces. That is, it is very hard to understand the structure of the user-interface by studying the code. This makes user-interface code hard to maintain.

Swul sets out to solve this problem by using a syntax that is closer to the conceptual idea of the Swing library. The central idea of Swul is that the implementation of a user-interface should reflect its hierarchical structure, i.e. subcomponents are subexpressions of their containers. They are not added afterwards in separate statements, which inevitably leads to tangling. Properties of components, such as widgets, containers, and layouts, can be set immediately on the component as well, thus defining all the aspects of a user-interface component at a single location.

However, the disadvantage of a separate DSL is that the integration with the rest of the program, written in a general-purpose language is cumbersome. Usually, escaping to the general-purpose language is restricted to certain places in the DSL and the connections between the domain-specific code and the general-purpose code are not verified by the compiler. For example, event handlers are often invoked by reflection if there is a separate user-interface specification.

```
menubar = {
  menu {
    text = "File"
    items = {
      menu item { text = "New"   accelerator = ctrl-N }
      menu item { text = "Save"  accelerator = ctrl-S }
    }}}

content = panel of border layout {
  center = scrollpane of textarea { rows = 20  columns = 40 }

  south = panel of border layout {
    east = panel of grid layout {
      row = {
        button of "Accept"
        button of "Cancel"
      }}}}
```

Fig. 2. Simple user-interface implemented in Java and Swul

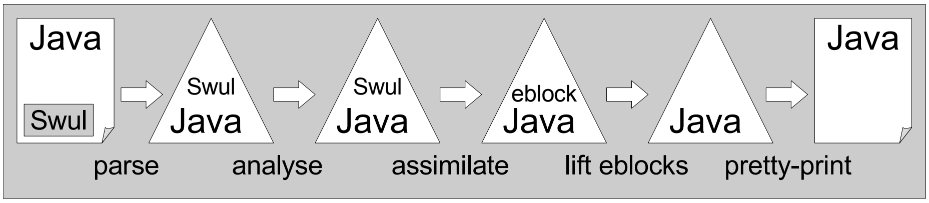

Fig. 3. Pipeline for the processing of Swul in Java

To integrate the user-interface, implemented in Swul, seamlessly with the rest of the program, we have embedded Swul in Java (JavaSwul). Swul components can be used as Java expressions (embed) and Java expressions can be used in place of Swul expressions (escape). For example, a custom border or component can be used in a Swul specification of a user-interface and event-handling code can be written in plain Java inside Swul. Figure 2 shows the implementation of the user-interface encoded in Figure 1 in Swul [2]. Here it is much easier to understand the structure of the user-interface, since this is directly reflected in the code. We will discuss the various aspects of Swul in more detail later.

Implementation Overview. A JavaSwul source file is processed by a series of components, which are of course available as a single tool to users of JavaSwul. The pipeline through which a source file is processed, is shown in Figure 3. The components will be discussed in the next few sections.

Note that the implementation, although it acts as a pre-processor to the Java compiler, is more solid than most pre-processors for several reasons. First, it operates on

[2] More examples are available at `http://www.strategoxt.org/JavaSwulExamples`

context-free syntax
```
  ComponentType Props?    -> Component {cons("Component")}
  "{" Prop* "}"           -> Props     {cons("Props")}
  PropType "=" PropValues -> Prop      {cons("Prop")}

  "{" Component* "}" -> PropValues {cons("PropMultiValue")}
  Component          -> PropValues {cons("PropSingleValue")}
```
context-free syntax
```
  "panel"           -> ComponentType {cons("JPanel")}
  "border" "layout" -> ComponentType {cons("BorderLayout")}
  "grid" "layout"   -> ComponentType {cons("GridLayout")}

  "content" -> PropType {cons("Content")}
  "layout"  -> PropType {cons("Layout")}
  "title"   -> PropType {cons("Title")}
```

Fig. 4. General productions of the Swul syntax definition

a complete abstract syntax tree, i.e. it is not based on lexical processing. Second, the pre-processor performs semantic analysis and type checking on the mixed AST. Hence, it is able to report semantic errors in terms of the original program. Most pre-processors only have knowledge of the lexical syntax and leave error reporting to the compiler. More advanced macro systems, such as [1, 3, 8], avoid lexical processing as well. For a discussion of the relation to macro systems see [6].

2.1 Syntax and Parsing

The syntactical part of the implementation of JavaSwul consists of a syntax definition for Swul itself and the embedding of Swul in Java. In all our embeddings we reuse an existing, modular syntax definition for Java 5.0.

Swul Syntax Definition. The syntax of Swul is defined in SDF, a modular language for syntax definition that integrates context-free and lexical syntax in a single formalism. Swul uses a combination of a general syntax and some sugar for specific circumstances. The most relevant productions from the syntax of the general syntax are shown in the first context-free syntax section of Figure 4. The general syntax is based on component types with the values of properties set between curly braces after the component type. Component types are for example `panel`, `button` etc. Examples of properties are `layout`, `text`, `horizontal gap`, and `border`. Some examples of component specific production rules of the syntax definition are shown in the second context-free syntax section of Figure 4.

In contrast to the general syntax of Swul presented here, the first edition of Swul [6] used component specific production rules and non-terminals. While extending the Swul language to cover more of the Swing library it became clear that this approach leads to a lot of duplication in the syntax definition, lots of non-terminals and, worse, poor error reports in case of syntactical errors in a JavaSwul source file. Therefore, we adopted this

```
context-free syntax
  (Modifier "-")* KeyEvent -> Prop {cons("Accelerator")}
  "ctrl"  -> Modifier {cons("CtrlModifier")}
  "alt"   -> Modifier {cons("AltModifier")}
  "shift" -> Modifier {cons("ShiftModifier")}
  "meta"  -> Modifier {cons("MetaModifier")}
```

Fig. 5. More domain-specific syntax in Swul

```
context-free syntax
  SwulComponent -> JavaExpr      {avoid, cons("ToExpr")}
  JavaExpr      -> SwulComponent {avoid, cons("FromExpr")}
```

Fig. 6. Syntactical Embedding of Swul in Java

general syntax and introduced a separate analysis phase that checks if the component types and properties are used in the right way.

Swul also supports user-friendly syntax for some domain-specific concepts that are hard to construct using the Swing API. For example, Swul introduces a concise notation for accelerator keys (key combinations to access a user-interface component with the keyboard). Figure 5 shows the SDF production rules for accelerators.

Keywords. The keywords used in the Swul production rules, such as `panel` and `border`, are not automatically *reserved* keywords. In general, reserved keywords are only necessary if ambiguities arise, for example between the keyword `null` and the identifier `null`. Moreover, if a separate scanner is used, then the scanner-parser combination cannot handle tokens that have different meanings in different contexts, i.e. if there is not interaction between the scanner and the parser. However, the META-BORG method is based on *scannerless* generalized-LR parsing, which can determine the meaning of a token based on the context in which it occurs, since there is no separate scanner. Thus, reserving these keywords (i.e. disallowing them as identifiers) is not required. However, they can still be declared as reserved keywords if this is desirable.

Embedding. The syntactic embedding of Swul in Java is defined in an SDF module that imports the Swul and Java syntax and defines where Swul components can be used in Java and vice versa. This embedding is defined by two productions, which are shown in Figure 6. The two production rules of this embedding define that a Swul component can be used as a Java expression (also known as a quotation) and that a Java expression can be used in Swul as a component (also known as escape or anti-quotation). Note that the embedding is a strictly modular combination of Java and Swul: we do not have to modify the Java or Swul syntax definition, thus we do not need to know the details of these syntax definitions either.

Renaming. To avoid unintended mixing of Swul and Java code, the non-terminals of the two languages have to be unique. Therefore, the embedding module imports SDF modules that prefixes all the Java and Swul non-terminals with the prefixes `Swul` and

```
module Java-15-Prefixed
imports Java-15
          [ CompilationUnit => JavaCompilationUnit
            TypeDec         => JavaTypeDec
            PackageDec      => JavaPackageDec
            ...
            Expr            => JavaExpr ]
```

Fig. 7. Prefixing all non-terminals of Java

Java respectively. These renaming modules are generated from the syntax definitions of Java and Swul. Figure 7 illustrates such a generated renaming module for Java. This renaming module imports the Java syntax definition (`Java-15`) and renames all non-terminals in this syntax definition by prefixing them with `Java`.

Cyclic Derivations. The METABORG method does not require the use of quotation and anti-quotation symbols to separate the embedded domain-specific language from the code written in the host language. Indeed, we do not use a quotation and anti-quotation symbol in the embedding of Swul in Java. Nevertheless, a problem with the two production rules for embedding Swul in Java is that they lead to cyclic derivations: a Swul component can be an expression, an expression can be a Swul component, which can be an expression, and so on. Scannerless generalized-LR cannot handle cycle derivations, so we have to disallow a cyclic derivation in some way. In [10] a trick was presented to cut off such cyclic derivations by using an existing language construct for disambiguation: *priorities* [4]. Priorities allow a concise specification of derivations that should be removed from the parse table by the parser generator. Usually, priorities are used for declaring the priority and associativity of operators, but in fact they can be used to disallow any production as the child of another production.

The following priority declares that the production for the escape from Swul to Java can never be applied immediately below the production that allows Swul to be used as a Java expression. This effectively cuts off the cycle in the derivations, and does not reject any useful interaction of Swul and Java.

```
context-free priorities
  JavaExpr -> SwulComponent > SwulComponent -> JavaExpr
```

2.2 Semantic Analysis

In the previous section we described how the Swul language is syntactically embedded in Java using the modular syntax definition formalism SDF. From this embedding we can generate a parser. Parsing a JavaSwul program results in an abstract syntax tree that is a mixture of Java and Swul language constructs. Before the Swul constructs are translated to plain Java code, we need to make sure that the source file does not contain semantic errors. If these errors will not be detected until compilation of the plain Java code, then the user of JavaSwul will have to map this error report to the original source file, which is undesirable.

```
dryad-type-of :
  Component(ct , _, Some(ComponentProps(ps )) ) -> <swul-to-swing> ct
  where <map(type-attr; check-property(|ct ))> ps

properties-of :
  BorderLayout() -> [North(), South(), East(), West(), Center() | xs]
  where <properties-of> Layout() => xs
```

Fig. 8. Type checking of embedded Swul

Therefore, we need to perform semantic analysis of the source file. To this end, we extend a type checker for Java, which is written in Stratego, with support for typing Swul code and the connections between Java and Swul. For example, we have to check that the Swul components are used correctly in the surrounding Java code and that the used Swul properties exist for the subject components. The code for this property check is sketched in Figure 8. Here, the type checker is extended with a new type rule `dryad-type-of` [3] that checks if a Swul component is used correctly. In this case, the properties of the component are checked, where the `properties-of` strategy is used by `check-property` to retrieve the available properties of a component. Thus, the existing type checker for Java, which invokes `dryad-type-of` to type expressions, is extended with a new type rule for the domain-specific Swul extension. If the Swul expression cannot be typed, then this will be reported.

2.3 Assimilation

Assimilation transforms a program with embedded domain-specific code to a program in the plain host language, in this case Java. So, the assimilation of Swul transforms the embedded Swul code to the corresponding invocations of the Swing API. A typical assimilation implemented in Stratego consists of a set of rewrite rules and a traversal strategy that controls the application of these rewrite rules. For most of the Swul language constructs, the rewrite rules are straightforward mappings of the convenient Swul syntax to more involved Swing library calls. However, some Swul constructs, such as event handling, require a more advanced treatment in the assimilation, since the generated Java code in these cases is not just locally inserted, as we will explain later.

Traversal. The traversal used in the assimilation is shown in Figure 9. The strategy is a generic top-down traversal where some Java and Swul language constructs are given a special treatment. A generic traversal is very useful for implementing assimilations of languages embedded in Java, since Java contains many different constructs. Implementing a specific traversal for Java and the domain-specific language by hand would take a lot of code and time. In all of the Stratego code fragments of this paper, the *`italic`* identifiers indicate meta-level (Stratego) variables. The Stratego code also uses concrete object syntax for Java and Swul (between |[and]|).

In the main traversal strategy (`swul-assimilate`), the special cases are preferred over the generic traversal combinator `all(s)`, which applies the argument `s` to all the

[3] Dryad is the name of the package that contains the Java type checker.

```
swul-assimilate =
     class-declaration
  <+ class-initializer
  <+ class-method
  <+ swul-expression
  <+ all(swul-assimilate)

class-initializer :
  |[ static { bstm1* } ]| -> |[ static { bstm2* } ]|
  where {| FieldModifier
        : rules(FieldModifier :+ _ -> |[ static ]|)
        ; <swul-assimilate> bstm1* => bstm2*
        |}

swul-expression = ?ToExpr(<SwulAs-Component>)
```

Fig. 9. Traversal strategy for Swul assimilation

subterms of the current term. The preferred alternatives (e.g. `class-declaration`) implement a more specific traversal for the cases where a generic traversal is not sufficient. One of the specific cases is `class-initializer`, which is shown in Figure 9. This special case keeps track of the context in which the assimilation traversal currently is: non-static (instance method) or static (class method). In a static context, fields that are generated by the assimilation, for example for event-handling, have to use a `static` modifier and therefore we have to keep track of this context. The assimilation of Swul uses a dynamic rule `FieldModifier` for this purpose. In the static context of a class initializer, the set of `FieldModifier` rules is dynamically extended with a new rule that produces the `static` modifier. If the dynamic rule strategy `bagof-FieldModifier`, which applies all FieldModifier rules, is invoked, then all current modifiers will be produced and these can be used in a fresh field declaration.

Another special case, illustrated in Figure 9, is the strategy `swul-expression`, which handles the transition from Java to Swul. This strategy is applicable to a `ToExpr` term, which is the constructor attached to the embedding production in Figure 6. For this term, the `swul-expression` strategy switches the traversal to Swul mode by invoking the `SwulAs-Component` rewrite rule.

Assimilation Rules. Figure 10 illustrates a number of Swul assimilation rules. In the assimilation of Swul we use a small extension of Java, called an eblock, that allows the inclusion of block statements in expressions. The syntax for eblocks is `{| statements | expression |}`. The value of an eblock is the expression. The statements are lifted by a separate tool to the statement before the statement in which the eblock occurs. There are also alternative eblocks for lifting statement the context *after* and before *and* after the context of the current expression. This small extension of Java has proven to be very effective for introducing new variables or performing side-effects in pure rewrite rules that need to transform an expression-level construct to a Java expression.

We now return to the rewrite rules of Figure 10. The first rule shows a typical rewriting for a Swing widget. The rewrite rule is a simple translation of the Swul construct to

```
SwulAs-JButton :
  |[ button { ps* } ]|{x} -> |[ {| x = new JButton(); bstm* |x|} ]|
  where <map(SwulAs-JButtonProp(|x))> ps* => bstm*

SwulAs-JPanel :
  |[ panel of c ]|{x} -> |[ {| x = new JPanel(); x.setLayout(e); |x|} ]|
  where <SwulAs-LayoutManager(|x)> c => e

SwulAs-GridLayout(|x) :
 |[ grid layout {ps*} ]|{y} -> |[ {| y=new GridLayout(i,j); |y|bstm*|} ]|
  where <nr-of-rows> ps*    => i
      ; <nr-of-columns> ps* => j
      ; <map(SwulAs-LayoutProp(|x,y))> ps* => bstm*

SwulAs-LayoutProp(|x,y) :
  |[ horizontal gap = c ]| -> |[ y.setHgap(e); ]|
  where <SwulAs-Component> c => e
```

Fig. 10. Some rewrite rules for assimilating Swul to Java

invocations of the Swing library. Note that a pre-eblock is used to create the `JButton` and set the properties of it. The second and third rule illustrate the rewriting of panels with a specified layout and the handling of the grid layout. Note that Swul does not require a specification of the number of rows and columns in a grid layout, since this can be calculated by the assimilator from the number of components in the columns and rows. The fourth rule assimilates the setting of the horizontal gap between components of a layout manager. The identifier of the subject layout manager is passed a term argument to the rewrite rule.

However, not all assimilation rules are that straightforward. For example, consider the event handling support of Swul. An example menu bar defined in Swul is shown in Figure 11. The `action event` properties of the `menu item` can contain a list of arbitrary statements that have the scope of the class declaration in which the menu bar is defined. A sketch of the code after assimilation is shown in Figure 12. The event handling code has been moved to a fresh inner class and the standard `EventHandler` class of Java is used to invoked the method declared in this inner class. A single instance of the fresh inner class is created and declared as a field of the class `MenuEvent`. This non-local assimilation of embedded Swul code is beyond simple rewriting (and also beyond typical macro expansion). The non-local assimilation is implemented by collecting the non-locally generated code in dynamic rules and inserting it in the right place on the way back in the traversal.

Producing Java. After assimilation, the abstract syntax tree is a plain Java abstract syntax tree, except for the expression block extension. These can be removed by invoking a tool in the Java support package for Stratego/XT. After this, we have a pure Java abstract syntax tree that can be pretty-printed using a standard pretty-printer for Java. The resulting source file can now be compiled with an ordinary Java compiler. Ideally, this should not result in additional semantic errors, since the semantic analysis phase

```
class MenuEvent {
  static void newFileEvent() { ... }
  static void main(String[] ps) { ...
    menubar = {
      menu {
        text = "File"
        items = {
          menu item {
            text = "New"
            action event = { newFileEvent(); }
          }
          menu item {
            text = "Exit"
            action event = { System.exit(0); }
          } ...
```

Fig. 11. Swul event handling

```
class MenuEvent {
  private static ClassHandler_0 classHandler_0 = new ClassHandler_0();
  public static void newFileEvent() { ... }

  public static void main(String[] ps) { ...
    JMenuItem_0 = new JMenuItem();
    JMenuItem_0.setText("New");
    JMenuItem_0.addActionListener(
      EventHandler.create(..., ClassHandler_0, "ActionListener_0", ""));
    ... }

  public static class ClassHandler_0 {
    public void ActionListener_0(ActionEvent event) { newFileEvent(); }
    public void ActionListener_1(ActionEvent event) { System.exit(0); }
  }}
```

Fig. 12. Swul event handling after assimilation

has already performed a full type check of the source file. However, the Java Language Specification defines many semantic rules, of which many are not related to type checking. Some of these are not yet implemented, so there is no absolute guarantee that errors will not occur after pre-processing until we have a fully compliant front-end for Java.

3 Other Examples

We have implemented several large embeddings to gain experience with the METABORG method. For example, we have embedded Java, AspectJ, XML, ATerms, XPath, and regular expressions in Java. In this section we will give a brief overview of two of these embeddings: regular expressions in Java and Java in Java.

```
regex ipline = [/
    ( ( [0-1]?\d{1,2} \. ) | ( 2[0-4]\d \. ) | ( 25[0-5] \. ) ){3}
    ( ( [0-1]?\d{1,2}    ) | ( 2[0-4]\d    ) | ( 25[0-5]    ) )
  /] ;

if( input ~? ipline )
  System.out.println("Input is an ip-number.");
else
  System.out.println("Input is NOT an ip-number.");
```

Fig. 13. Regular expression syntax embedded in Java

JavaRegex. We have designed an extension of Java, called JavaRegex, for string matching and rewriting using regular expressions. The purpose of JavaRegex is to provide compile-time checking of the syntax of regular expressions and to introduce new, high-level operators specific to regular expressions and string processing. This extension makes regular expressions much easier to use in Java. Compared to Perl, which has such facilities built in the language, writing regular expressions in plain Java is cumbersome, since they have to be encoded in string literals. The regular expressions are first interpreted as strings and secondly as regular expressions, meaning that the programmer needs to deal with special characters in the first and in the second interpretation at the same time. This results in an escaping-hell, where even experienced regular expression users carefully have to count the number of escapes that are used. Furthermore, basic operations in string processing are often compositions of several method invocations of the standard Java regular expression library, which makes the library harder to use. Nevertheless, the basic functionality of the library is quite well designed, so we would only like to provide a different syntax to the operations provided by this library.

Figure 13 shows a basic application of JavaRegex. In this example, the basic features of JavaRegex are used: regular expression syntax (`[/ /]`), regular expression types (`regex`), and testing if a string matches a regular expression (`~?`). In the quotes of a regular expression there is no need to escape the special characters of Java, hence solving the escaping-hell by providing a literal regex context. Note that the regular expression syntax is easy to implement due to the use of scannerless parsing, since context-sensitive analysis of lexical syntax is supported by design. JavaRegex also supports named capture groups in a regular expression, where the names immediately refer to Java variables. Furthermore, JavaRegex provides *rewrites* as a more abstract operator for string processing. Rewritings can be composed using sequential and choice operators and can be used in string traversals.

The assimilation of JavaRegex translates the regular expressions to Java string literals and the operators to invocations of the standard Java library for regular expressions. The assimilation not only translates the JavaRegex extensions to straightforward API invocations, but also generates control-flow to deal with the rewriting extensions of JavaRegex. The assimilation acts as a pre-processor of the Java compiler, but, we would like to avoid that the user of JavaRegex gets compiler errors in terms of the generated Java code, which would be hard to track down in the original source file.

```
dryad-type-of :
  ToBooleanExpr(x,y) -> Boolean()
  where <type-attr> x => TypeString()
      ; <type-attr> y => Regex()

dryad-type-of :
  Assign(x, y) -> Regex()
  where <type-attr> x => Regex()
      ; <type-attr> y => Regex()
```

Fig. 14. Regular expression syntax embedded in Java

Therefore, the assimilation phase performs semantic analysis of the source file. For this, we have have extended a type checker for Java with type checking rules for the JavaRegex extensions. The type checker is based on abstract interpretation, where each expression rewrites to its type. Thus, rewrite rules are added that rewrite the JavaRegex extensions to their types and check the types of the arguments. Figure 14 shows the rules for matching `~?` (`ToBooleanExpr`) and regex assignments (`Assign`). This last rule extends the existing type checking rule for assignments. This shows that the type checking of existing language constructs can be extended in a modular way using rewrite rules.

JavaJava. A common problem in the embedding of domain-specific languages are ambiguities. The ambiguities can arise between different constructs of the domain-specific language or between the host language and the domain-specific language. In particular, this is a problem in meta-programming with concrete object syntax [12], where quotations and anti-quotations usually have to be disambiguated explicitly by indicating the non-terminal of the quotation (e.g. Jak, which is part of of the JTS/AHEAD Tool Suite [2]). In a meta-language with a manifest type system this explicit disambiguation is redundant.

In [5] we present a meta and object language independent method for solving the ambiguity problem in meta-programming with concrete object syntax. The method uses scannerless generalized-LR parsing to parse meta-programs that use concrete object syntax. This produces a forest of all possible parses. An extension of a type-checker for the host language disambiguates the forest to a single tree by removing alternatives that cannot be typed. If more than one alternative can be typed, then an ambiguity is reported. This method of disambiguation extends the METABORG method by providing a *reusable* tool for disambiguating programs that use an embedded domain-specific language. Indeed, this tool is also useful for the disambiguation of the embedding of Swul in Java. This method of disambiguation generalizes the language-specific and not reusable approach of, for example, Meta-AspectJ [14], where explicit disambiguation is not necessary either.

We have used this method to embed AspectJ (similar to [14]) and Java in Java for generative programming without requiring explicit disambiguation. The implementation of the embedding of Java in Java consists of assimilation rules that translate embedded Java 5.0 abstract syntax to the Eclipse JDT Core DOM. Figure 15 shows an example of a JavaJava program. The quotation in this program (between |[and]|) is ambiguous.

```
CompilationUnit dec;
String x = "y";
dec = |[ public class Foo {
           public int bar() {
             return #[x] * x;
           }} ]|;
```

Fig. 15. Java embedded in Java witout explicit disambiguation

For example, the code in the quotation can represent a full compilation unit, a single type declaration, or a list of type declarations. The scannerless generalized-LR parser will produce all these possible parses. Next, the type checker will eliminate the alternatives that cannot be typed, leaving only the compilation unit alternative, since *dec* is declared as a compilation unit in this program. JavaJava also supports anti-quotations (`#[]`), which are disambiguated in a similar way. The second x in the quotation represents a meta-variable (a variable in the meta program) and is inserted in the resulting abstract syntax tree without requiring an explicit anti-quotation.

Note that the implementation of disambiguation is independent of the embedding of Java in Java and can therefore be used for the disambiguation of other ambiguous embeddings of domain-specific languages.

4 Conclusion

We have presented examples and an overview of the METABORG method for introducing embedded domain-specific syntax to overcome the lack of abstraction in general-purpose languages. We have presented three different examples of the embedding of a domain-specific language, designed syntactically and semantically for three different application domains: user-interfaces (JavaSwul), string processing (JavaRegex), and code generation (JavaJava). These examples illustrate that modular syntax definition and scannerless generalized-LR parsing are excellent tools for syntactically embedding a domain-specific language in a general-purpose host language. Furthermore, we have shown how Stratego's rewrite rules, traversal strategies, and dynamic rules can be applied to concisely assimilate the embedded code to the host language. Also, we have sketched how a type checker for the host language can be extended to support semantic analysis of the combination of the host language and the domain-specific language.

Availability. Stratego/XT and the Java support packages are Free Software (LGPL) and available from `www.strategoxt.org`. More information on METABORG is available at `www.metaborg.org`, where you can find references to related publications and applications (including JavaBorg, JavaSwul and JavaJava).

Acknowledgments. We want to thank Rob Vermaas for his extensive contribution to the development of JavaJava. We thank the reviewers for their detailed feedback.

References

1. J. Baker and W. Hsieh. Maya: multiple-dispatch syntax extension in java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281. ACM Press, 2002.
2. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse (ICSR'98)*, pages 143–153. IEEE Computer Society, June 1998.
3. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'02)*, pages 31–40. ACM Press, 2002.
4. M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
5. M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *LNCS*, pages 157–172, Tallinn, Estonia, September 2005. Springer.
6. M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.
7. C. Gould, Z. Su, and P. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, Washington, DC, USA, 2004. IEEE Computer Society.
8. B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, November 1966.
9. Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.
10. J. J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, November 2005.
11. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
12. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
13. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.
14. D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In G. Karsai and E. Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, volume 3286 of *LNCS*, pages 1–19, Vancouver, Canada, October 2004. Springer.