MetaBorg: Domain-specific Language Embedding and Assimilation

Aivar Annamaa
aivarannamaa@hotmail.com

November 2009

Abstract

It is recognized that providing programmers with languages that are close to problem domain can increase productivity significantly. Creating a new full-blown language can be too expensive, therefore domain specific languages (DSL-s) are usually combined with a general purpose language (GPL). Generative programming has been shown to be useful for this kind of language extensions.

This report focuses on technologies behind MetaBorg, a complete generative programming solution proposed by Eelco Visser and his team. It provides means for combining syntaxes and semantics of a base GPL and DSL-s. Using the definition of combined language it creates tools for translating programs written in new language back to the base language.

1 Introduction

Solutions for complex problems tend to come in shape of large and complex programs, yet when source code is examined, usually only small part of the code deals directly with the problem, much of the effort goes into writing some kind of boilerplate code. It is not only a problem of writing, noise in source code affects understanding the solution even more. Separation of generic code into libraries and frameworks can help, but often it's still too cumbersome to express some domain problems only in terms of classes or other abstraction devices fixed by the base language.

Combining general purpose languages with different domain-specific languages is not a new idea – SQL and regular expressions have been around for long time – but in recent years many authors have proposed that using DSL-s could be even more common practice, if the cost of creating new languages and respective tooling (IDE-s, debuggers etc.) were smaller [6, 9, 11]. One approach for

achieving this would be developing systems, that generate necessary tools from high-level language descriptions.

In the following sections we give a bottom-up overview of technologies behind MetaBorg. MetaBorg is a collective name for set of tools and techniques for embedding and assimilating DSL-s into some GPL. Here, assimilation means translating DSL fragments to the base language, this enables programmer to continue using most of the existing infrastructure for base language. Basically, MetaBorg method enables building preprocessor systems, that parse combined language, transform abstract syntax tree and unparse resulting tree back to text.

First we look at the problems of parsing embedded languages. For defining grammars, a domain specific language is used, called SDF (Syntax Definition Formalism). We describe the language and it's usage, focusing on aspects relevant to combining languages - disambiguation, modularity and scannerless parsing.

For translating fragments of embedded language into base language, MetaBorg method uses Stratego/XT, a program transformation language and toolset. We focus on it's specialty – fine grained control over term rewriting using rewriting strategies. We also see how term rewriting can be used for semantic analysis (eg. type checking).

In order to make results from parsing or transformations usable, Stratego toolkit provides means for generating and combining relevant command-line tools.

We then gather these topics together and give an example of MetaBorg method in use – we describe how Swul (a language for defining user interfaces) is embedded and assimilated into Java.

Finally we review most important technological aspects of MetaBorg and compare the system with other similar solutions.

2 Combining grammars with SDF

Combining syntaxes of two languages can be challenging, but it's still possible to do it by hand. But in order to mix more than two languages or to create a more generic solution, where languages to be mixed in are not known in advance, more structured approach is needed.

MetaBorg uses Syntax Definition Formalism (SDF) for defining and combining syntaxes and SGLR (Scannerless Generalized LR parser) for parsing the resulting syntax. Following section first gives overview of SDF and then describes some difficulties in parsing embedded languages and shows how certain distinctive features of SDF and SGLR overcome these problems [4].

SDF is a domain-specific language for defining syntaxes of context-free languages. It supports the entire class of context-free grammars, including ambiguous grammars. It favors clean definition of the language over the hacks needed to eliminate ambiguities [1] – for this it provides special disambiguation facilities discussed below. In order to achieve greater reusability, SDF syntax definitions can be split into modules.

It's possible to express both lexical and context-free syntax in SDF. Lexical syntax section corresponds roughly to traditional scanner definition, but it's not restricted to regular grammar. Context-free section corresponds to traditional parser definition. The terms lexical and context-free syntaxes in SDF do not refer to different expressiveness – both sections can describe context-free languages. The difference lies in handling the whitespace – in lexical section whitespace between symbols has to be shown explicitly, in context-free section whitespace is assumed implicitly.

Lexical syntax sections usually define constructs like identifiers and literals. Also, the definition of whitespace should be given here – it's marked by special non-terminal symbol `LAYOUT`.

```
lexical syntax
  [A-Za-z][A-Za-z0-9]* -> Id
  [0-9]+               -> IntConst
  [\r\n\t\ ]           -> LAYOUT
```

Symbols defined in lexical section are used as building blocks in the context-free section for defining more complex non-terminals. Note that definition of expressions is straightforward, without extra productions for handling priority, associativity or ambiguities.

```
context-free syntax
  Id -> Exp {cons("Var")}
  IntConst -> Exp {cons("Int")}
  Exp "+" Exp -> Exp {cons("Plus")}
  Exp "-" Exp -> Exp {cons("Min")}
  Exp "*" Exp -> Exp {cons("Mul")}
  Exp "/" Exp -> Exp {cons("Div")}
  Exp "." Id "(" {Exp ","}* ")" -> Exp {cons("Call")}
  "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
```

Productions are annotated with a constructor name (eg. "Plus"). First a parse tree is constructed and this is transformed to abstract syntax tree using given constructors. Resulting AST can be presented using ATerm format [7]. For example, the input `4 + x.get(5)` is represented by the following ATerm:

[1] Often ambiguities are resolved by introducing new productions, but unfourtunately this compilicates the grammar

```
Plus(Int("4"), Call(Var("x"), "get", [Int("5")]))
```

2.1 Disambiguation

SDF definition are used together with Scannerless Generalized LR parser (SGLR). Like any GLR parser, it can work with ambiguous grammars (returning parse forest instead of parse tree) but in practice usually a single parse tree is needed. SDF contains facilities to reject certain parse trees. The example syntax definition above has several ambiguities. First, there is *associativity* problem in binary operators. SDF allows specifying associativity as annotation in production rule, for example

```
Exp "+" Exp -> Exp {left, cons("Plus")}
```

Another problem is *priority* of operators (and also method calls). In SDF priorities are defined relatively instead of priority levels. Relative priorities (eg. `+` vs. `*`) deal with several production rules, therefore priority specifications are not given together with productions but in a separate section. In following example three priority groups are shown, separated by priority operator `>`

```
context-free priorities
    Exp "." Id "(" {Exp ","}* ")" -> Exp
  > {left:
      Exp "/" Exp -> Exp
      Exp "*" Exp -> Exp }
  > {left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp }
```

There is still one problem left with the example grammar – in SDF the policy of longest match is not followed by default and therefore erroneous input `if 1 then aelse b` would be valid. This can be solved by explicitly defining *follow restrictions*. A follow restriction `A -/- CC` specifies, that non-terminal `A` can't be followed by a character from character class `CC`

```
lexical restrictions
  Id                -/- [A-Za-z0-9]
  IntConst          -/- [0-9]
  "if" "then" "else" -/- [A-Za-z0-9]
```

2.2 Modularity

In order to make syntaxes of base language and embedded language maintainable, both languages should be defined separately. It's even more important when different combinations of base language and DSL-s are considered. For example, if we want to combine Java, XML, SQL, XPath and regular expressions, then providing a single syntax for all these it would be unacceptable from the point of view of clarity, maintenance and reusability.

Usually syntax definition techniques and corresponding parser generators limit grammars to some subset of context-free grammars. This is usually not a problem when designing a single language, but when combining different grammars, the resulting grammar normally doesn't fit to any common subclass of context-free grammars (like LALR, or LL). Therefore, in order to fully support language embedding, SDF supports full class of context-free grammars.

Subsection 5.1 gives an example of two different grammars defined in separate modules and joined together.

2.3 Scannerless parsing

Most parsers don't use text as input, but work on token stream prepared by separate scanner. Scanner's job is to identify different chunks of input as literals, names, keywords or separators. Usually languages are designed so, that scanners can be specified using regular grammars and are therefore relatively easy to implement. This scheme of dumb scanner separated from smart parser brings restrictions to language design, for example it disallows nested multiline comments (`/* /* */ */`).

When combining languages, especially when languages are very different, it becomes more difficult to identify the tokens and assign correct type to them – the same piece of text can, for example, represent a keyword in one language and identifier in other language. When scanner and parser are united, then there is always more context information available for determining the type of current token. That is why SDF is designed to be used with a special kind of GLR parser – Scannerless GLR parser [4].

There is another problem with separate scanners – if a token (eg. string literal) has some inner structure (eg. escape sequences), then scanner ignores this and hands the token over to parser as one opaque chunk. If a string literal contains a piece of program in a DSL, then this structure needs to be parsed in runtime. Scannerless parsing can deal with this in the context of main language parsing and therefore can make language embedding easier. For example, the following XML attribute contains an escape to the meta level

```
<a href="http://www.<% s %>.org">Stratego/XT</a>
```

In the embedding of XML in Java, SDF and SGLR could present the attribute as:

```
Attribute(
  QName(None, "href")
, DoubleQuoted([
    Literal("http://www.")
  , Literal(FromExpr(ExprName(Id("s"))))
  , Literal(".org")
  ])
)
```

3 Transformation with Stratego/XT

After the program written in mixed syntax is parsed, the DSL parts need to be translated into base language, so that resulting source code can be fed into standard base language compiler or interpreter. Assimilation is defined by set of rewriting rules and strategies (together these are called transformation rules) written in Stratego language [10, 3]. This definition gives semantics to DSL in terms of base language. Besides translating code from one language to another, program transformation techniques are also used for type-checking.

In this section we describe program transformation facilities of Stratego/XT in general and in section 5 we give specific example of using transformations for language assimilation.

3.1 Rewrite rules

Basic transformation of terms are defined by *rewrite rules*. A rule has the form `L: l -> r`, where `L` is the label of the rule, `l` is term pattern to be matched and `r` is resulting term pattern. Term pattern can be a variable or a constructor applied to zero or more term patterns. So, term pattern is a term with variables in it.

```
ReShape: Plus(a, Plus(b, c)) -> Plus(Plus(a, b), c)
```

There can be several rule definitions with same label, ie. one rewrite rule can consist of several cases.

A conditional rewrite rule adds a computation, that should succeed in order for the rule to apply:

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add>(i, j) => k
```

Here `<add>(i, j) => k` is a shorthand notation for applying a *strategy* (`add`) (strategies are discussed in section 3.3) to a term (`(i, j)`) and matching the result against another term (`k`) [3]. Basically, a rule is applied to a term when it's left hand side and condition can be unified with the term.

3.2 Concrete Object Syntax

When rules include larger program fragments as patterns, then using ATerm format (abstract syntax) can be too inconvenient. Therefore Stratego allows patterns to be written in concrete syntax of the object language. As an example, the EvalPlus rule from above can be written like this:

```
EvalPlus : |[ i + j ]| -> |[ k ]| where <add>(i, j) => k
```

The use of concrete syntax is indicated by *quotation delimiters* `|[` and `]|`.

Let's consider a larger program fragment that implements instrumentation of method calls and would be quite cumbersome in ATerm format. Following rule modifies function definitions to call some (logging) functions before and after original function body:

```
TraceProcedure :
  |[ function f(x) = e ]| ->
  |[ function f(x) =
   (enterfun(String(f)); e; exitfun(String(f))) ]|
```

Basically, the example is written in a language which is an extension to Stratego language – fragments of a specific programming language embedded into a host language. Thus, Stratego implementation itself also deals with issues described in this report – embedding and assimilating languages.

The concrete syntax version has all properties of the abstract syntax version: pattern matching, term structure, it can be traversed, etc. In short, the concrete syntax is just syntactic sugar for the abstract syntax. In the example above, symbols `f`, `x`, `e` and `String` are Stratego *meta-variables*, not tokens of object language. Here the distinction between meta-variables and other tokens is implicit and is based on specific SDF constructs used in the definition of object language[3]. Following example shows how to make escaping to the meta level explicit with *anti-quotation* operator `~`

```
TraceProcedure :
  |[ function ~f(~x) = ~e ]| ->
  |[ function ~f(~x) =
    (enterfun(~String(f)); ~e; exitfun(~String(f))) ]|
```

3.3 Transformation strategies

Most rewriting engines apply rewriting rules exhaustively, ie. rules are applied to a term and it's subterms until no further matches are found. This is not always desirable (some rewriting systems are not confluent or terminating), therefore in Stratego one has to define rewriting strategy explicitly.

There are several commonly used strategies, that could be provided as built in primitives (like exhaustive innermost or outermost normalization, single pass top-down etc.), but instead of that, Stratego provides user with combinators for easy composition of strategies. With these combinators, one can create generic strategies, that can be further parameterized with the set of rules. This separation of rules and strategies allows better reuse of both.

Basically, a strategy is an algorithm, that transforms a term into another term or fails at doing so. Strategies are composed using the following combinators:

- sequential composition (`s1; s2`)
- deterministic choice (`s1 <+ s2`), here first `s1` is tried, if this fails then `s2` is tried
- non-deterministic choice (`s1 + s2`); same as previous, but order of trying is not specified, this is used when two rules are mutually exclusive
- testing (`where(s)`), ignores the transformation achieved, effect can be in rejecting the parent transformation
- `not(s)`, reverse of previous ie. transformation succeeds if `s` fails

Strategies can be labeled, just like rules, using strategy definitions. Following example defines a strategy named `repeat` which repeats strategy `s` until it fails.

```
repeat(s) = try(s; repeat(s))
```

Note that strategy definitions do not mention the term to which they are applied - this is not needed, as all strategies take a term as input and give another term as output.

3.4 Term Traversal

So far we have shown how to apply rewrite rules to the root node of the term. In order to apply different strategies to different subterms, the term must be traversed. Most important mechanism for term traversal in Stratego is called *congruence operators*. For each constructor C, a corresponding congruence operator exists with the same name. This operator defines a strategy $C(s_1, \ldots, s_n)$

that only applies to terms with constructor C and arity n. When matched, each substrategy s_i is applied to respective subterm. With this operator different strategies can be selected for different subterms. Here is an example strategy, that specifies identity strategy (ie. no transformation) for certain subterms of certain types of terms:

```
control-flow(s) = Assign(id, s) + If(s, id, id) + While(s, id)
```

As congruence operators mention specific constructors, they are specific to a datatype. Stratego also provides combinators for creating generic traversals. Operator `all(s)` applies strategy `s` to each direct subterm of the current term. Following example shows how `all` can be used for defining strategy for bottom-up traversal

```
bottomup(s) = all(bottomup(s)); s
```

3.5 First-Class Pattern Matching

So far we have made distinction between strategies and rewrite rules. However, neither of them are primitive constructs- Rewrite rules are just syntactic sugar for simple strategies composed from more basic transformation actions like matching and building terms and delimiting the scope of pattern variables. These basic actions can be directly used to form some interesting matching idioms.

Let's consider the rewrite process of an example rewrite rule introduced in section 3.1:

```
EvalPlus : Plus(Int(i), Int(j)) -> Int(k) where <add> (i, j) => k
```

First subject term is matched against the pattern in the left-hand side and variables `i` and `j` are bound, then `where` condition strategy is evaluated and variable `k` is bound, finally right-hand side pattern `Int(k)` is instantiated and it replaces the original term. The scope of variables `i`, `j` and `k` is the rule.

In following example, these separate actions are expressed explicitly using primitive actions for matching (`?pat`), building (`!pat`), and scoping ({$x_1, \ldots, x_n$`:s`}). In fact, that's how Stratego compiler desugars the definitions.

```
EvalPlus =
  {i,j,k: ?Plus(Int(i), Int(j)); where(!(i,j); add; ?k); !Int(k)}
```

The strategy `where(s)` succeeds if strategy `s` succeeds, but it returns the original subject term [3].

3.6 Scoped Dynamic Rewrite Rules

Normal rewrite rules can only use information obtained by pattern matching on the subject term. Many useful transformations would also need information from the context of program fragment in focus. Example uses would be bound variable renaming, typechecking, constant propagation, common-subexpression elimination, dead code elimination. To achieve this, Stratego extends strategies with *scoped dynamic rewrite rules* [10, 3].

Dynamic rules are generated at run-time and can access information from their generation contexts. Consider following example strategy, meant for inlining single-argument functions:

```
DeclareFun =
  ?|[ function f(x) = e1 ]|
  ; rules(
      InlineFun : |[ f(e2 ) ]| -> |[ let var x := e2 in e1 end ]|
    )
```

At runtime, each time a function definition is matched and strategy `DeclareFun` is applied, `rules(...)` construct creates a new dynamic rule (named here as `InlineFun`) to be used at call sites of this specific function. Declaring `InlineFun` in the scope of match to function definition makes `f` in the left-hand side and `x` and `e1` on the right-hand side refer to inherited bindings of these variables. The storage and retrieval of the context information is handled transparently by Stratego and is of no concern to the programmer.

Here is a more complex example of inlining that applies to functions with arbitrary arity:

```
DeclareFun =
  ?fdec@|[ function f(x1*) ta = e1 ]|;
  rules(
    InlineFun :
      |[ f(a*) ]| -> |[ let d* in e2 end ]|
      where <rename>fdec => |[ function f(x2*) ta = e2 ]|
          ; <zip(BindVar)>(x2*, a*) => d*
  )
BindVar :
  (FArg |[ x ta ]|, e) -> |[ var x ta := e ]|
```

Construct `fdec@` on second line is used to name a pattern.

Dynamic rules are first-class entities and can be used at global level of term traversal. It's possible to control the application of dynamic rules using rule

scopes. For example, `DeclareFun` and `InlineFun` as defined above, could be used in the following inlining strategy (notation `{|L : s|}` restricts the scope of a generated rule `L` to the strategy `s`):

```
inline = {| InlineFun :
              try(DeclareFun)
            ; repeat(InlineFun + Simplify)
            ; all(inline)
            ; repeat(Simplify)
          |}
```

First inlining rules are generated for all functions encountered by `DeclareFun`, then function calls are inlined using `InlineFun` and some simplifications are made. Next, strategy applied recursively to subterms using `all`. Finally, on the way up, simplification is applied again. Of course, actual inliner should also decide whether function should be inlined at all.

3.7 Term Annotations and Type Checking

A term consists of a constructor and argument terms. In Stratego a term has also a list of annotations, so far the terms in our examples just happened to have empty annotation lists. A term with annotations has the form `t{a1, ..., am}`, where each a_i is a term. Annotations can be added during parsing (eg. `cons` annotations shown in section 2) or, since annotations are also terms, they can be pattern-matched and rewriting rules and strategies can be applied to them. Therefore it's possible to create transformations that add type annotations to terms – that's how Stratego can be used for type checking. Following example shows how to assign type annotation to an expression by matching type annotations of it's components:

```
TypeCheck : Plus(e1{Int}, e2{Int}) -> Plus(e1, e2){Int}
```

Whole typing/typechecking process may include assigning some types during parsing (eg. to literals), then applying set of typing rules and finally checking that resulting root node has a type annotation.

Similarly many other program analyses can be expressed as program transformation problems. Actual examples in which annotations were used include escaping variables analysis in a compiler for an imperative language, strictness analysis for lazy functional programs, and bound-unbound variables analysis for Stratego itself [10, 3].

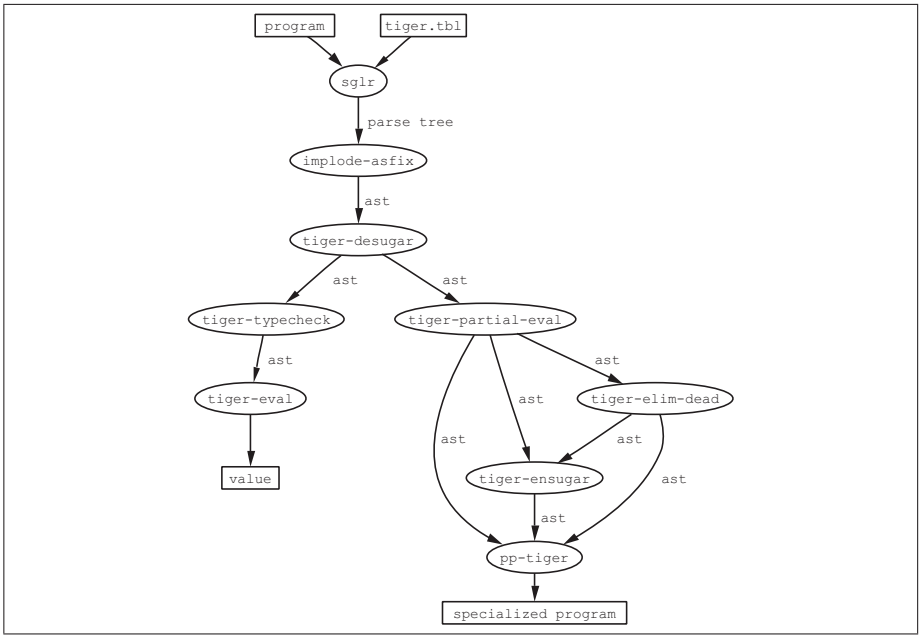

Figure 1: Example data flow between tools

4 Infrastructure in Stratego/XT

Stratego language together with XT toolkit is able to generate standalone tools from grammar definition and transformations. Figure 1 shows the data flow between the tools composed to form the interpreter and partial evaluator for Tiger language.

Some of the tools are generic and are included in XT toolset (there are more than 100 available), other tools are user-written. Following example shows how to turn `simplify` strategy into a standalone tool:

```
module simplify
imports lib Tiger-Simplify
strategies
  main = io-wrap(simplify-options, simplify)
  simplify-options =
    ArgOption("-O", where(<set-config> ("-O", <id>)),
              !"-O n     Set optimization level (1 default)")
```

Stratego compiler first produces C code and then compiles it to binaries using

```
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"
    }}}}
```

Figure 2: Simple UI implemented in Swul

standard C compiler. The compiler and required libraries are available under LGPL license.

Tools can be combined using pipes, whereas tree data is represented using ATerm format [7]. Both plain-text and optimized binary representation of ATerm can be used. As there are ATerm libraries available for several languages, some components of a transformation system can be written in languages other than Stratego.

Finally, after transformations, AST is turned into text again (unparsing). For this, Stratego uses Generic Pretty-Printer (GPP) package[5]. With GPP, a tree is first transformed to an intermediate representation in *Box* language, which coqntains text together with markup related to pretty printing. This can processed further by different back ends to produce, for example, plain-text, LaTeX, HTML, etc.

5 Example: Swul in Java

Swul is a DSL for writing graphical user-interfaces using Swing library [2]. It aims to make structure of the user interface and relationships between components clearly visible from the source code. Figure 2 shows an example program fragment written in Swul.

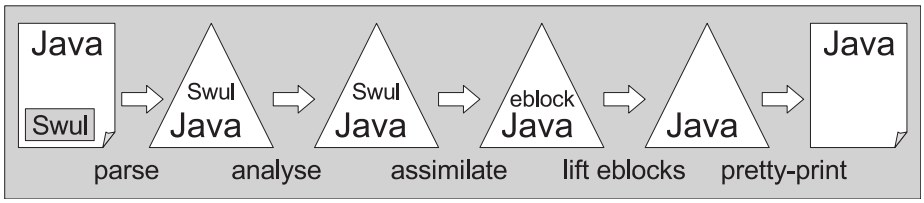

Figure 3: Pipeline for the processing of Swul in Java

With MetaBorg method, Swul components can be used inside Java code in place of Java expressions (embedding) and Java expressions can be used in place of Swul expressions (escaping).

Figure 3 shows stages of processing for Java-embedded Swul code. Basically, MetaBorg works like a preprocessor to Java compiler, but unlike most preprocessors, it operates on the complete (Java+Swul) abstract syntax tree. Secondly, it performs semantic analysis and type checking on mixed AST, so it is able to report semantic errors in terms of the original program.

5.1 Syntax Definition and Embedding

For creating such a preprocessor, first syntax for Swul was defined using SDF (Java 1.5 syntax is available as a SDF package).

Syntactic embedding of Swul and Java is defined in a separate SDF module that imports both syntaxes and defines where Swul components can be used in Java and vice versa:

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

Here `SwulComponent` and `JavaExpr` are nonterminals from Swul and Java grammars, respectively, and new productions in combined grammar allow using one nonterminal in place of the other.

To avoid name collisions, both syntaxes are imported through automatically generated renaming modules, where symbol names are prefixed with language names (eg. `JavaExpr` instead of `Expr`).

MetaBorg method does not require the use of quotation or anti-quotation symbols for changing between the languages (this enables to keep the combined syntax cleaner), but this raises issues of cyclic derivations. In SDF this ambiguity problem is solved by priority declaration

```
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")}
```

Figure 4: Core productions of Swul grammar

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

From the combined syntaxes of Java and Swul, a SGLR parsing table is compiled and this can be used to create standalone parser for the new language.

5.2 Semantic Analysis

When programs written in the combined language are parsed, then resulting AST needs to be checked for semantic errors (for example type errors). It is possible to ignore error checking until embedded language constructs are translated away to base language, but in this case error messages are created in terms of translated program and mapping problems back to original code can be too difficult. Therefore semantic analysis is performed on AST with mixed language constructs.

In case of Swul, type checking is based on a previously available Java type checker, written in Stratego. This is extended to support typing Swul code and connections between Swul and Java.

5.3 Assimilation

Assimilation of Swul transforms the embedded Swul code to the corresponding invocations of the Swing API. For this, a set of transformation rules are needed together with a traversal strategy. Following fragment shows main traversal strategy used for assimilating Swul:

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

Basically it is a list of strategies to choose from when translating some term. Note that the special cases are listed first and generic traversal combinator `all(s)` (applying strategy `s` to all subterms) comes last.

One of the specific cases is `class-initializer`, which keeps track of whether current context is an instance method or class method (static). In a static context, new generated fields need to be declared static as well and for this, the set of `FieldModifier` rules is extended with a new dynamic rule that produces the static modifier:

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

Most assimilation rules are quite straightforward, for example rewriting Swul expression to a Swing widget (semantics of construction `{| s | e |}` is described below):

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

This example demonstrates use of an *eblock*, a convenience extension to Java, that allows the inclusion of block statements in expressions. The syntax for eblock is `{| statements | expression |}`, the value of eblock is the expression. In this example, the term to be matched is an expression, but its translation introduces also some statements (`x = new JButton (); bstm*`) that

should actually be somewhere before the expression in focus. Without eblocks, this transformation would need more complex nonlocal transformation rule. Before producing final Java code, eblocks are transformed to normal set of statements using a separate tool.

Finally, when AST is transformed to plain Java, it is pretty-printed to source code. There are pretty printers for Java and some other languages included in Stratego/XT toolkit. The resulting file can be compiled with standard Java compiler. Ideally, this should not generate error messages, because type checking was already done on mixed language AST.

6 Conclusion and Related Work

This report described MetaBorg – a method for embedding and assimilating domain-specific languages into a general purpose language. This task requires solving many technical difficulties.

For creating grammar of new combined language, specific features of SDF (Syntax Definition Formalization) are used. Modular definitions allow defining grammars separately and combine them without restrictions. As new grammar usually does not fall into some well-known subclass of context-free languages, a powerful parsing technology – Scannerless GLR – is used.

Assimilation consists of transforming DSL fragments into host language constructs. MetaBorg builds on strengths of Stratego program transformation language – transformation strategies, scoped dynamic rewrite rules and term annotations. Usage of ATerm format for data exchange allows creating small reusable tools for common program transformation tasks.

There are other approaches for extending languages with new constructs. Among these, this LISP macros are probably best known, but they operate in much simpler environment (s-expressions) than, for example, Java syntax. Another interesting solution is *Metamorphic Syntax Macros* [1] – an extendable language using specific BNF-like statements in program source-code. This is different from MetaBorg mainly because it uses a specific base language, and proposed extension techniques cannot be applied to arbitrary languages. JetBrains' *Meta Programming System* [6] and Language Workbench from Intentional Software [9] are both platforms for creating, combining and using languages. They avoid concrete syntax issues by storing programs in tree form and they enable advanced program editing and browsing using customizable views. Rascal [8] is a language and toolset quite similar to Stratego/XT but it includes also some extra facilities for program analysis.

To conclude the comparison – MetaBorg method enables extending any existing language, using special purpose languages for syntax definition and transformations and the resulting source code is usable with standard compilers.

References

[1] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *In Proceedings of Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2002. ACM*, pages 31–40. ACM Press, 2000.

[2] Martin Bravenboer, René De Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05*. Springer Verlag, 2005.

[3] Martin Bravenboer, Trygve Karl Kalleberg, and Eelco Visser. Stratego/XT tutorial. http://strategoxt.org/Stratego/StrategoDocumentation.

[4] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM.

[5] M. de Jonge. A pretty-printer for every occasion, 2000.

[6] Sergey Dmitriev. Language oriented programming, 2004.

[7] H. A. De Jong, P. Klint, P. A. Olivier, Issn x, Mathematisch Centrum (smc, The Dutch Foundation, Mark G.J. van den Brand, Hayco De Jong, Paul Klint, and Pieter Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30:2000, 2000.

[8] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: a domain specific language for source code analysis and manipulation. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, 2009. to appear.

[9] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 451–464, New York, NY, USA, 2006. ACM.

[10] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems, 2004.

[11] Martin P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15(4):147–161, 1994.