# Higher-order Transformations with Nested Concrete Syntax

Rob Economopoulos
ECS, University of Southampton
Southampton, SO17 1BJ, UK
gre@ecs.soton.ac.uk

Bernd Fischer
ECS, University of Southampton
Southampton, SO17 1BJ, UK
b.fischer@ecs.soton.ac.uk

## ABSTRACT

Transformations play an important role in grammar-based applications such as program generation. In this domain, the use of the concrete syntax technology is particularly beneficial as it substantially simplifies the development and maintenance of the transformations. Further benefits could be achieved by the use of higher-order transformations to generate program transformations. However, both technologies cannot be combined easily because of the difficulties in merging the different object, meta, and meta-meta languages. Here we propose an approach to higher-order transformations with nested concrete syntax. We use Stratego as meta-meta language and allow the embedding of arbitrary object languages into arbitrary meta languages. We describe the implementation of the approach and give two examples for its application, the embedding of Stratego in itself to generate WebDSL program transformations, and the use of Stratego to generate Prolog-clauses with embedded object syntax.

## Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal definitions and theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Translator writing systems and compiler generators*

## General Terms

Languages

## Keywords

Code generation, Program transformation

## 1. INTRODUCTION

Transformations are everywhere in grammar-based applications and tools: for example, program transformations are often used to implement program generators. However, the development of transformation systems remains difficult [7]. Such systems are often specified as a set of rewrite rules [6, 2, 5], where the rules are formulated over the terms of an abstract syntax such as prefix constructor terms or XML. For "small" languages, this notation is concise and adequate, but for languages with a large grammar this becomes cumbersome very quickly [11]. The concrete syntax technology [11, 3] mitigates these problems. It allows developers to use the syntax of the target language (or *object language*) to formulate the object structures in the rewrite rules, i.e., in the *meta language*. Meta variables designate positions where dynamically constructed object-level terms are injected at run-time. The combined language can be parsed with a parser generated from both language definitions that also handles the quotation and unquotation operators. The resulting abstract syntax trees (ASTs) contain a mixture of meta and object language syntax; the embedded object language fragments are *assimilated* or *exploded* [4] into pure meta language representation that can then be processed by the tools of the meta language.

Independently, transformation development can also be simplified by *higher-order transformations* (HOTs), i.e., transformations with another transformation as input or output. These are becoming common in model-driven engineering (cf. [9]), but less so in grammar-based applications. One reason might be that it is now even more difficult to express object program fragments; in particular, they need to be expressed in the meta-meta language's abstract syntax representation of the meta language's abstract syntax.

Unfortunately, both technologies (i.e., concrete syntax and higher-order transformations) cannot be combined easily, because of the difficulties in merging the object, meta, and meta-meta languages, which can all be different. The main problem is that we can now escape from the embedded object language into two different language levels, and can thus inject object-level terms constructed at two different stages, the run-time of the higher-order transformation, and the run-time of the generated transformation, respectively. The explosion mechanism needs to be aware of both the level on which the escape occurs and the level to which it refers, in order to build the right ASTs. At the same time, we can also escape from the meta language to the meta-meta language. This situation makes it difficult to combine explosion mechanisms that have been developed independently.

Here we thus propose a general approach to higher-order transformations with nested concrete syntax. We use the Stratego [5] transformation language as meta-meta language and allow the embedding of arbitrary object languages into arbitrary meta languages. We describe the implementation of the approach and give two examples for its application, the embedding of Stratego in itself to generate transformations of WebDSL programs, and the embedding of Prolog-clauses with embedded object syntax into Stratego rules. We use the Stratego-Stratego-WebDSL combination to extend the WebDSL code generator indirectly, by systematically changing the generated code.

```
generate-java-servlet-page :
    def |[ define mod* x_page(farg*) req* { elem* } ]| ->
    <emit-java-code-local> |[
        package pkgname;
        import java.util.*;
        import javax.servlet.*;
        public class x_page extends PageServlet {
          public String getPageName() {
            return "~x_page";
        } }
    ]|
```

**Figure 1: Stratego rewrite rule using concrete syntax to generate a Java servlet page for a WebDSL page definition.**

```
module EmbeddedWebDsl[M]
imports
 WebDSL       // object language syntax
exports
 context-free syntax
  "webdsl"  "|[" Application      "]|" -> M{cons("ToTerm")}
  "webdsl"  "|[" TemplateElement  "]|" -> M{cons("ToTerm")}
  "webdsl*" "|[" TemplateElement* "]|" -> M{cons("ToTerm")}
  "def"     "|[" Definition       "]|" -> M{cons("ToTerm")}
  "def*"    "|[" Definition*      "]|" -> M{cons("ToTerm")}
 context-free syntax
  "~app:"  M       -> Application      {cons("FromTerm")}
  "~templElem:" M  -> TemplateElement  {cons("FromTerm")}
  "~templElem*:" M -> TemplateElement* {cons("FromTerm")}
  "~def:"  M       -> Definition       {cons("FromTerm")}
  "~def*:" M       -> Definition*      {cons("FromTerm")}
 variables
  "x_"[A-Za-z0-9]* -> Id {prefer}
```

**Figure 2: SDF for embedding of WebDSL in meta-language at non-terminal M.**

```
module EmbeddedWebDSLMix[Ctx M]
imports EmbeddedWebDSL[M]
        [ Application       => Application[[Ctx]]
          TemplateElement   => TemplateElement[[Ctx]]
          Definition        => Definition[[Ctx]] ]

module Stratego-WebDSL-Java
imports
 StrategoMix[Host]
 EmbeddedWebDslMix[WebDSL Term[[Host]]]
 languages/java/EmbeddedJavaMix[Java Term[[Host]]]
hiddens
 context-free start-symbols Module[[Host]]
```

**Figure 3: SDF for embedding of WebDSL and Java in Stratego.**

## 2. BACKGROUND

### 2.1 Stratego

Stratego is a general transformation language that is centered around four different themes: (*i*) *rewrite rules* to express basic transformations, (*ii*) programmable *strategies* to guide the application of these rules, (*iii*) *concrete syntax* to express the patterns of rules in the syntax of the object language, and (*iv*) *dynamic rewrite rules* to express context-sensitive transformations [5]. It is complemented by the XT framework, a collection of tools that provide grammar engineering and parsing support [5].

In Stratego, rewrites rule have the form $R : p_1 \rightarrow p_2$ `where` $s$, where $R$ is the rule name, $p_1$ resp. $p_2$ are patterns, or terms with variables, and the optional trailing `where`-clause specifies a condition $s$ that restricts the applicability of the rule. $s$ is evaluated after $p_1$ is matched but before $p_2$ is rebuilt. Terms are built from constructors defined by means of a signature giving the respective argument types and the result type for each constructor. Stratego is built on top of SDF [10], which is used to define an external syntax for the terms and to derive the signature from the parse rules.

Stratego uses programmable strategies to determine how a term is traversed, and how and where rules are applied. The application of a single rewrite rule is a simple strategy that transforms a term at the root. This strategy fails (similar to the notion of failure in Prolog) when the left-hand side does not match or the condition fails. More complex strategies are built up using strategy combinators and definitions. The fundamental combinators are sequential composition $s_1$ `;` $s_2$, and deterministic choice $s_1$ `<+` $s_2$, which first tries to apply strategy $s_1$ and if that fails, applies $s_2$. One-level traversal combinators such as `all` apply a strategy to the direct subterm(s) of a term; more complex traversals are then defined recursively.

### 2.2 Concrete Syntax in Stratego

Since rewrite rules over abstract syntax are large and difficult to read [11], Stratego allows an object program's concrete syntax to be used instead. Figure 1 shows the rule *generate-java-servlet-page* that matches the concrete syntax of a WebDSL [13] page definition and transforms it into a (partial) Java servlet page. The concrete syntax appears between the `|[` and `]|` quotation brackets and meta-variables (e.g., *`x_page`*) are typeset in italics. Identifiers in front of the quotation brackets denote the syntactic category of the embedded fragment, which is sometimes required for disambiguation.

In order to embed an object language's concrete syntax into Stratego it is necessary to combine its SDF syntax definition with that of Stratego, and to define the actual embedding, i.e., the syntax of the quotation and unquotation operators as well as the meta-variables. The `EmbeddedWebDsl` module shown in Figure 2 extends the imported object language (i.e., WebDSL) by the quotation and unquotation operators; by convention, their abstract syntax uses the constructors `ToTerm` and `FromTerm`. The module parameter `M` specifies the non-terminal symbol of the—yet unspecified—meta language at which the embedding happens. Such *embedding* modules must be defined manually for each embedded language; however, these modules are very regular, and we omit the definition of the corresponding `EmbeddedJava`.

The syntax definitions of language embeddings are re-usable for multiple host languages, provided unintentional embeddings are prevented. This is achieved via grammar mixins [4], which rename all non-terminals of a language embedding with the *context* of the host language in which the embedding is used. The renaming exploits SDF's *parameterized sorts* (e.g., `Definition[[Ctx]]`). The mixins can be generated automatically by the `gen-sdf-mix` tool, and the first module in Figure 3 shows the mixin corresponding to the `EmbeddedWebDSL` fragment in Figure 2. The final module in Figure 3 imports the syntax for Stratego, WebDSL and Java via the grammar mixins and defines a new start-symbol for the language combination. From this combination, a parser can be automatically generated that recognises and builds ASTs of Stratego programs with concrete syntax patterns.

Before the combined ASTs can be processed by the Stratego compiler, the embedded object language fragments (i.e., WebDSL and Java) must be transformed into pure Stratego. This *assimilation*, or *explosion*, to Stratego can be done generically for any object language as long as the embeddings to and from Stratego are labeled consistently. The combined AST is traversed top-down, and any `ToTerm`-node (which indicates that its argument is in the WebDSL or Java abstract syntax), is exploded into Stratego abstract syntax. Conversely, any `FromTerm`-node in this subtree indicates

```
application Simple
  imports initializeDB
  entity User {
    uname  :: String (name)
    pword  :: Secret
  }
  section pages
    define page root() {
      header{"Welcome"}
      table {
        for (u:User) {
          row {column{output(u.name)}
               column{navigate(editUser(u)){"edit"} }
    } } }
    define page editUser(u:User){
      form{
        input(u.name)
        input(u.pword)
        submit action{u.save();} {"save"}
    } }
```

**Figure 4: Simple WebDSL application.**

a fragment of (escaped) Stratego abstract syntax, so no transformation is required. Implicit meta-variables are used directly as Stratego variables.

The combined language parser and assimilator are actually part of the Stratego compiler, so all that is required to use concrete syntax patterns in Stratego programs is the combined language syntax definition that uses the `ToTerm` and `FromTerm` constructors to mark the transition to and from Stratego.

## 2.3 WebDSL

WebDSL is a high-level, domain-specific language for dynamic web applications with rich data models [13]. Figure 4 shows a small WebDSL application that stores a group of users and displays them in a page. At its core is the data model, here just comprising the `User` entity definition with properties `uname` and `pword` that have the types `String` and `Secret`, respectively. The structure and layout of the web pages is defined through WebDSL's presentation sub-language but the actual formatting is handled by existing tools such as CSS. Here, the root `page` contains the `header` output construct followed by a `table` with all users. Control flow statements such as `if` and `for` loops can be used to dynamically define the structure of a page. WebDSL also provides abstraction mechanisms such as templates to avoid code duplication.

The WebDSL implementation uses code generation by model transformation [12]. The generator normalizes the WebDSL ASTs (via several transformation steps defined in Stratego) to a core representation that is close to the target platform, but which still contains domain-specific concepts. The architecture of the WebDSL code generator follows the classical four-layer architecture [1] of OMG.

## 3. EMBEDDING STRATEGO RULES WITH CONCRETE SYNTAX IN STRATEGO

In Section 2.2, we sketched the embedding of WebDSL in Stratego; however, the Stratego compiler can only handle the embedding of one level of concrete syntax in a meta-language. This makes it impossible to write higher-order transformations with concrete syntax. In the following, we thus describe an extension of the approach that allows the embedding of Stratego's concrete syntax in itself, i.e., we use Stratego as both the meta and meta-meta language of the higher-order transformation. We focus in particular on extending the existing (single-level) embedding of WebDSL into Stratego. The purpose of the nested embedding is to simplify the formulation of Stratego transformations that can modify WebDSL applications. We use this Stratego-Stratego-WebDSL combination to add an access control sublanguage to the WebDSL core language. We show in Section 4 how the approach can be applied to other language combinations, where the meta-meta and meta languages are different.

## 3.1 Access control in WebDSL

Access control (AC) is essential for web applications as it restricts which data users can see and which operations they can execute. WebDSL uses an integrated AC sublanguage to declare AC policies on pages, templates and actions directly. The language is policy neutral so that any AC policy can be defined and implemented, including discretionary, mandatory, and role-based AC policies [14].

```
principal is User with credentials uname, pword
access control rules {
  rule page root(*) { true }
  rule page editUser(u:User) { principal == u }
  rule action *(*) { true }
}
```

Above is a simple WebDSL AC definition for the web application shown in Figure 4. It places no restrictions on access to the `root` page, or any of the actions in the application, but limits the access to the `editUser` page—only users logged in can edit their own data. The WebDSL code generator desugars these AC specifications into conditionals and weaves them into the code for controlled pages, templates, or actions before the normalization stage.

## 3.2 Language Embeddings

As an experiment in language and generator extension, we duplicated the existing AC functionality in WebDSL and then extended it with a simple fine-grained AC, although the latter will not be detailed here. We extracted the AC syntax definition from the implementation of WebDSL and turned it into a standalone language (with SDF definition `Ac`). The same policies that can be defined in WebDSL can now be defined separately from WebDSL applications. We then implemented higher-order transformations (described in detail in Sect. 3.3) that in effect realize a separate generator for the AC language that uses core WebDSL as target language. Sect. 3.4 describes how the extension can be integrated with the existing core generator.

The nested embedding in Figure 5 follows the same general approach as the simple embedding outlined in Sect. 2.2. `StrategoStrategoWebDslAc` imports the host (Stratego) and guest languages (Stratego, AC, WebDSL), again via grammar-mixins so that no name space conflicts occur between the combined syntax definitions. The Stratego grammar is imported twice, but in two different versions, reflecting Stratego's dual role: `StrategoMix` provides access to Stratego as meta-meta language while `EmbeddedStrategoMix` provides access to Stratego as meta language, so that Stratego concrete syntax fragments can be used in the patterns of the transformation system. `EmbeddedStratego` therefore extends Stratego with the quotation and unquotation operators, as well as the syntax of Stratego meta-variables, similar to the `EmbeddedWebDsl` module shown in Figure 2.

The major difference is in the handling of the object language, i.e., WebDSL. Since we need to be able to escape from WebDSL into two different levels of surrounding Stratego (i.e., into the meta-meta and meta levels, resp.), we need to define two different embeddings. The first embedding handles the interaction between object level and meta level. It defines the quotation operators as in Figure 2,injecting them into the non-terminal symbol `M` of the

```
module EmbeddedWebDsl2[M MM]
imports
 WebDSL       // object language syntax
exports
 context-free syntax
  "webdsl"  "|[" Application      "]|" -> M{cons("ToTerm")}
  "webdsl"  "|[" TemplateElement  "]|" -> M{cons("ToTerm")}
  "webdsl*" "|[" TemplateElement* "]|" -> M{cons("ToTerm")}
  "def"     "|[" Definition       "]|" -> M{cons("ToTerm")}
  "def*"    "|[" Definition*      "]|" -> M{cons("ToTerm")}
 context-free syntax
  "~app:"  Emb        -> Application      {cons("FromTerm")}
  "~templElem:" Emb  -> TemplateElement  {cons("FromTerm")}
  "~templElem*:" Emb -> TemplateElement* {cons("FromTerm")}
  "~def:"  Emb        -> Definition       {cons("FromTerm")}
  "~def*:" Emb        -> Definition*      {cons("FromTerm")}
  M                   -> Emb              {cons("FromTerm")}
 variables
  "x_"[A-Za-z0-9]+ -> Id {prefer}
 context-free syntax
  "~~app:" MM         -> Application      {cons("FromTerm")}
  "~~templElem:" MM  -> TemplateElement  {cons("FromTerm")}
  "~~templElem*:" MM -> TemplateElement* {cons("FromTerm")}
  "~~def:" MM         -> Definition       {cons("FromTerm")}
  "~~def*:" MM        -> Definition*      {cons("FromTerm")}

module StrategoStrategoWebDslAc
imports
 StrategoMix[Host]
 EmbeddedAcMix[AcOjb Term[[Host]]]
 EmbeddedStrategoMix[StratObj PreTerm[[Host]] Term[[Host]]]
 EmbeddedWebDsl2Mix[WebDslObj Term[[StratObj]] Term[[Host]]]
hiddens
 context-free start-symbols Module[[Host]]
```

**Figure 5: SDF for nested embedding of WebDSL and Stratego in Stratego.**

host language (i.e., Stratego as meta language) given as module parameter. We also use a chain rule that enforces the use of two `FromTerm` constructors to represent the doubly-nested origin of the unquotations and thus to ensure the right representation on the meta-meta level. The second embedding handles the interaction between object level and meta-meta level using the second parameter of the module. Since we do not allow WebDSL fragments on the meta-meta level, it only defines the unquotation operators; we use a different operator (˜˜) than in the single nested case to distinguish them syntactically.

Although both `EmbeddedStrategoMix` and `EmbeddedWebDSL2Mix` are parameterized with three non-terminal symbols, only the first parameter of each module (i.e., the context of the host language in which the embedding is used) perform the same function. The second and third parameters of the `EmbeddedStrategoMix` module are an artifact of the way the Stratego syntax is defined.

## 3.3 Higher-Order Transformations with Embedded WebDSL

Using the nested concrete syntax as described above, it is straightforward to define a HOT that implements a compiler for the AC sub-language by rewriting the generated core WebDSL code. Figure 6 shows the Stratego implementation. The `Hook` module contains the strategies and rewrite rules required to generate an executable Stratego program and a collection of auxiliary transformations for WebDSL. It provides a framework needed to generate the HOT and can be re-used to implement any HOT for WebDSL. The `generate-ac-rules` module contains the WebDSL transformations proper.

Recall that WebDSL applications are defined using high level abstractions that are normalized down to a core representation by the generator. Since restrictions may need to be applied to core constructs that are automatically generated during this process, our transformations must be defined on the core representation. Hence, the concrete syntax patterns used to match WebDSL code fragments that we want to transform must also be written in the core representation. However, this requires a detailed knowledge of the WebDSL generator and the way it normalizes high level constructs, which is not straightforward. Ideally, an extensible generator should expose the normalizations it applies, or better yet, supply the functionality to perform such normalizations; however, the WebDSL generator does not do this. We have instead extracted and reverse-engineered the required normalization strategies for the constructs used in our transformation from the generator's source code. The transformation of a `navigate` element to it's core representation is shown at the bottom of Figure 6. More normalizing transformations can be added by redefining the `desugar` rewrite rule, which is applied to the generated Stratego program by the `do-boilerplate` strategy.

Another consequence of basing our transformations on the core representation is that we also need to deal with the renaming of identifiers done by the generator. Ideally, an extensible generator should supply a renaming mapping, but again the WebDSL generator does not do this. However, the core representation includes annotations containing the original identifiers, and we have used this to re-construct the required mapping in form of a dynamic rule. This dynamic rule is used by the rewrite rule generated by `RestrictTemplate`.

The main idea of the HOT is to generate a Stratego module called `RewriteAc` (cf. rule `createModule`) for each AC specification as shown in Section 3.1. This contains a number of strategies for each specified access control rule. Each strategy simply wraps an access control test around the code corresponding to the controlled WebDSL element. This module is then compiled (using the standard Stratego compiler) and spliced into the transformation pipeline of the WebDSL compiler.

## 3.4 Extending the WebDSL Code Generator

The `RewriteAc` module generated by applying the HOT shown in Figure 6 to an AC specification as shown in Section 3.1 contains a *customization patch*, i.e., strategies that rewrite core WebDSL. Such patches must be applied to the independently generated core WebDSL programs in order to weave in the AC functionality. Fortunately, we can apply the patches and integrate the AC without modifying (or even re-compiling) the WebDSL code generator at all. WebDSL provides an option to output the intermediate representation at every stage of the generation process, including the final core representation. Because this is syntactically valid WebDSL, we can modify and feed it back into the WebDSL code generator and generate the Java for the application, re-using the existing backend. We use this technique to extract the core representation and implement a separate transformation tool in Stratego to apply customization patches. The diagram in Figure 7 describes this process. Note that we can rely almost entirely on components provided or generated by the XT framework; however, we need a small sed script to fix minor syntactic inconsistencies in the pretty-printed text.

## 3.5 Evaluation

Extending Stratego to allow the embedding of concrete syntax in itself was relatively straightforward. The major difficulty was to get the assimilation to handle nested `FromTerms` correctly. The HOTs that implement the AC language are (in our opinion) concise and clear and can be seen as an operational semantics of the

```
module generate-rules
imports libstratego-lib Hook Ac Stratego
strategies
  generate-ac-rules = ?AcDef(<alltd(RestrictTemplate <+ RestrictPage)>)
                      ; <do-boilerplate(|"RewriteAc")>
rules
  RestrictTemplate :
    newacrule|[ rule template x_id(margs) {e_check acrule*} ]| ->
    Def|[
      Modify : elem|[ define template ~id:X_ID2(~farg*:FARG*) {~elem*:ELEM*} ]| ->
               elem|[ define template ~id:X_ID2(~farg*:FARG*) {
                         if (~~exp:e_check) {"Access denied"} else {~elem*:ELEM*}} ]|
      where namesMatch(|~NoAnnoList(Str(<double-quote> x_id)), X_ID2)
    ]|
  RestrictPage :
    newacrule|[ rule page x_id(margs) {e_check acrule*} ]| ->
    Def |[
      Modify : elem|[ define page ~~id:x_id(~farg*:FARG*) {~elem*:ELEM*} ]| ->
               elem|[ define page ~~id:x_id(~farg*:FARG*) {
                         if (~~exp:e_check) {"Access denied"} else {~elem*:ELEM*}} ]|

      Modify : elem|[ navigate(~~id:x_id(~exp*:EXP*)){~elem*:ELEM*} ]| ->
               elem|[ if (!checkAccess(~exp*:EXP*)) {"Access Denied"}
                       else {navigate(~~id:x_id(~exp*:EXPS)){~elem*:ELEM*}} ]|
    ]|

module Hook
imports libstratego-lib Stratego
strategies
  do-boilerplate(|MODULENAME) =
    ?RULES
  ; !Module|[ module MODULENAME
              imports libstratego-lib libwebdsl-front strategies
              strategies
                main = io-wrap(alltd(GetNameMap);alltd(Modify))
                GetNameMap = ?TemplDef@Define([Template()]
                               , NAME{OriginalNameAnno(ORIGNAME{ANNO*})}
                               , ARGS
                               , None()
                               , BODY*
                              )
                ; origName := <strip-annos>ORIGNAME
                ; newName  := <strip-annos>NAME
                ; rules(OrigNameMap : origName -> newName)
                ; !TemplDef
               namesMatch(|origName, matchedName) =
                 <OrigNameMap> origName => x_id
               ; x_id := <strip-annos> matchedName
              rules
                 ~*RULES
    ]|
  ; top-down(try(desugar))
rules
  desugar : NavigateCall(PageCall(x,arg*),y*,z*) ->
    TemplateCall("navigate",[ThisCall(x,arg*)],y*,TemplateBody(z*))
```

**Figure 6: HOT to generate Stratego program that will transform WebDSL to include specified customized access control rules.**

AC language. In principle, they were easy to formulate; in practice, however, it took some effort to formulate them in such a way that they were applied uniformly, due to the renaming and desugaring issues described above. We have reverse engineered these steps from the WebDSL compiler and factored out our implementation into a boilerplate "hook" module, but we believe that such hook modules should (and indeed easily could) be provided by the generator developers, in order to facilitate extensions and external modifications. Moreover, hook modules are only required if the HOTs are applied to some internal representation but formulated in terms of the original representation. In our case, we chose this approach to ensure that the generated AC rules also apply to templates and pages automatically generated by WebDSL, without the need to write a large number of special cases.

As an alternative to HOTs with Stratego as meta language we could also use Stratego's dynamic rules, which phase-shift the construction of the rules to the runtime, and thus do not need the meta-meta level and its quotations. However, dynamic rules are difficult to inspect, while the result of the HOT can easily be inspected at the source level. Moreover, dynamic rules require the same hooks as HOTs (or even worse, a tight integration with the object language compiler), and, of course, must be supported by the meta language.

## 4. GENERALIZING THE APPROACH

Our approach can also be applied in situations in which the meta-meta and meta languages are different, and the use of dynamic rules is not an option. Here, we present the implementation of a HOT that uses Stratego to generate Prolog clauses with embedded concrete syntax, in this case propositional infix operators, such as /\ and \/. This scenario is deliberately simple, and only intended to demonstrate the generality of the approach; however, we have already used the integration of a more complex concrete syntax into

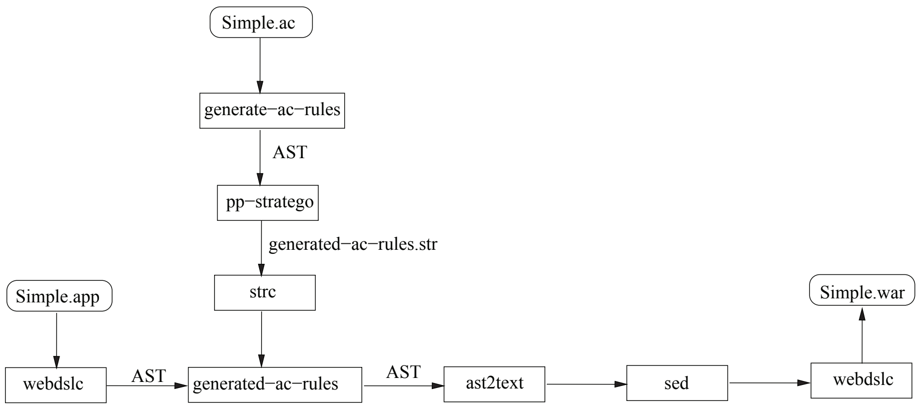

**Figure 7: Pipeline of customization patch generation and application to generated code.**

```
imports Prolog
exports
 context-free syntax
           "|[" Program "]|" -> M {cons("ToTerm")}
  "prog"    "|[" Program "]|" -> M {cons("ToTerm")}
  "clause"  "|[" Clause "]|"  -> M {cons("ToTerm")}
  "clause*" "|[" Clause* "]|" -> M {cons("ToTerm")}
 context-free syntax
  "$var:" M         -> Variable    {cons("FromTerm")}
  "$name:" M        -> Name        {cons("FromTerm")}
  "$word:" M        -> Word        {cons("FromTerm")}
  "$qname:" M       -> QuotedName  {cons("FromTerm")}
  "$t:" M           -> Term        {cons("FromTerm")}
  "$t*:" M          -> {Term ","}+ {cons("FromTerm")}
  "$c:" M           -> Clause      {cons("FromTerm")}
  "$c*:" M          -> Clause*     {cons("FromTerm")}

module EmbeddedProp2[MM M]
imports Prop
exports
 context-free syntax
         "|[" Prop "]|" -> M {cons("ToTerm")}
  "prop" "|[" Prop "]|" -> M {cons("ToTerm")}
 context-free syntax
  "$prop:" Emb          -> Prop {cons("FromTerm")}
  "$id:"   Emb          -> Id   {cons("FromTerm")}
  "$id*:"  Emb          -> Id*  {cons("FromTerm")}
  M                     -> Emb  {cons("FromTerm")}
 context-free syntax
  "$$prop:" MM          -> Prop {cons("FromTerm")}
  "$$id:" MM            -> Id   {cons("FromTerm")}
  "$$id*:" MM           -> Id*  {cons("FromTerm")}

module StrategoPrologProp
imports
 StrategoMix[Host]
 EmbeddedPropMix[PropObj Term[[Host]]]
 EmbeddedPrologMix[PrologObj Term[[Host]]]
 EmbeddedProp2Mix[PropObj2 Term[[Host]] Term[[PrologObj]]]
exports
 context-free start-symbols Module[[Host]]
```

**Figure 8: SDF for nested embedding of Prop and Prolog in Stratego.**

Prolog in previous work [8], so the scenario is not overly simplistic.

## 4.1 Using Stratego to Generate Prolog with Concrete Syntax

The SDF in Figure 8 shows the embedding of Prop into Prolog that is then in turn embedded in Stratego. Its structure is very similar to that shown in Section 3.4. `StrategoMix` adds the syntax of Stratego to the language combination, while the grammar mixins of Prolog and Prop enable the use of their concrete syntax in Stratego. The first parameter of all three modules is used as the language context of the combination to avoid any name space conflicts between the combined languages. The second (parameterized) non-terminals used in the above imports are the meta-language non-terminals that the concrete syntax fragments should be injected into, and escaped to. `EmbeddedPropMix` has a structure similar to `EmbeddedPrologMix`, so we have not included it here. It is added to the language combination to so that Stratego strategies can be used to manipulate Prop expressions that are not embedded in Prolog.

The `EmbeddedProp2Mix` mixin defines the embedding of Prop in Prolog and adds the ability to escape into both the surrounding Prolog concrete syntax and the outer Stratego. The mixin is again generated by the `gen-sdf-mix` tool, using the two-level embedding `EmbeddedProp2`. The mixin is again parameterized with three non-terminals, where the first defines the context it is used in, and the other two are the original embedding parameters, i.e., the meta-meta (Stratego) non-terminal and meta-language (Prolog) non-terminals that can be escaped to, and into which the Prop concrete syntax fragments should be injected, resp. The `$` and `$$` symbols are defined as the unquotation operators to indicate an escape to Prolog and Stratego, respectively. Since the syntax rules defining the injections of concrete syntax fragments to and from the meta-language require the `ToTerm` and `FromTerm` abstract syntax constructors, the syntax rules defining the escape to Prolog again use a chain rule to enforce that the abstract syntax constructed for the escape sequence is surrounded by *two* FromTerm constructors. This ensures that the assimilation works correctly and is explained in detail in the next section. The (artificial) example in Figure 9 demonstrates how the embedding of Prolog with nested concrete syntax into Stratego can be used. The Stratego rule `hot1` rewrites Prolog clauses which use propositional expressions in their body. Note that `A` is a propositional variable, not a Prolog variable. The generated transformation replaces this by the result of the strategy `hot2` (here simply a term representing `true`), which is evaluated when `HOTtest` is compiled. Also note that the escaped meta variable `X` is not bound anywhere, which means that it remains a free Prolog variable, whereas `P` can have a value when the predicate `a` is called.

## 4.2 Exploding Nested Embedded Concrete Syntax Generically

The combined syntax definition of meta and embedded languages plays a key role in Stratego transformation systems that use concrete syntax patterns. The parser, automatically generated from

```
module HOTtest
imports Prolog Prop metaSig libstrategolib
rules
 hot1 :
  clause |[b(P,Prop) :-
           Prop = |[(A /\ $prop:X) \/ $prop:P]|.
         ]| ->
  clause |[a(P,Prop) :-
           Prop = |[($$prop:<hot2> /\ $prop:X) \/ $prop:P]|.
         ]|
strategies
  hot2 = !prop |[true]|
```

**Figure 9: HOT for the Prolog-Prop language.**

this definition, constructs ASTs that contain a mixture of the different abstract syntaxes. These ASTs need to be assimilated or exploded into pure Stratego abstract syntax that can be processed by the Stratego compiler. Stratego's existing assimilator can unfortunately not handle language embeddings that have more than one level of nested concrete syntax, such as the embedding described in the previous section. We have thus extended it to deal with arbitrarily deeply nested concrete syntax fragments. The core of the new assimilator is shown in Figure 10, with the extensions highlighted.

The main strategy of Stratego's assimilator is `MetaExplode`, which traverses the term until it finds a `ToTerm` subterm. This indicates the beginning of an embedded language subterm that needs to be exploded into pure Stratego abstract syntax. The strategy `trm-explode` recognizes and handles escaped subterms (i.e., meta variables and unquoted fragments), and recursively deconstructs the terms of the embedded language (cf. `TrmOp`). However, it only handles a single level of embedded code: to extend to multiple embedding levels, we extended it with the `ToTermPatchFromTerm` strategy that matches embedded `ToTerm` constructors, indicating the beginning of another embedded language fragment.

The subterms of the embedded `ToTerm` are exploded by the new `trm-explode2` strategy, which is similar to the original *trm-explode* strategy, but which handles escaping into different language levels. Code to be spliced from a different language level is marked by `FromTerm` constructors. A single `FromTerm` refers to the outer (i.e., meta-meta) language level, in this case Stratego. Two nested `FromTerm` constructors refer to the second (i.e., meta) language level, in this case Prolog. When a `FromTerm` is matched by `TrmFromTerm2`, it is removed and its subterms are translated into Stratego abstract syntax. Any nested `FromTerms` that remain must refer to an embedded meta-language, and need to appear in the generated Stratego code, so they get translated to the form `Op("FromTerm",`$a_n$`)`, where $a_n$ are the exploded subterms. Note that our new assimilator does not depend on any of the embedded languages. All that is required is that any embedded code is marked with (nested) `FromTerm` constructors that are defined in the SDF language embedding.

Figure 11 shows as example a combined Stratego-Prolog-Prop AST corresponding to the *rules*-section of the HOT in Figure 9. The terms in normal font are Stratego abstract syntax, the terms in bold font Prolog abstract syntax, and the terms in the light-coloured font are Prop abstract syntax. The first `ToTerm` indicates the beginning of an embedded Prolog fragment

```
|[b(P,Prop):-Prop=|[(A /\ $prop:X) \/ $prop:P]|.]|.
```

The second `ToTerm` marks the beginning of a Prop fragment

```
|[(A /\ $prop:X) \/ $prop:P]|
```

and the nested `FromTerms` mark Prolog that must be spliced in.

```
Rules(
  [ RDefNoArgs(
      "hot1"
    , RuleNoCond(
        ToTerm(
          NonUnitClause(
            Func(
              Functor(Word("b"))
            , [Var("P"), Var("Prop")]
            )
          , BodyGoal(
              Infix(
                Var("Prop")
              , Op(Symbol("="))
              , ToTerm(or(and(
                            var("A")
                          , FromTerm(FromTerm(Var("X"))))
                       , FromTerm(FromTerm(Var("P"))))
                ))))
        )
      , ToTerm(
          NonUnitClause(
            Func(
              Functor(Word("a"))
            , [Var("P"), Var("Prop")]
            )
          , BodyGoal(
              Infix(
                Var("Prop")
              , Op(Symbol("="))
              , ToTerm(or(and(
                 FromTerm(RootApp(CallNoArgs(SVar("hot2"))))
                , FromTerm(FromTerm(Var("X"))))
              , FromTerm(FromTerm(Var("P"))))
                ))))
      ))))])
```

**Figure 11: StrategoPrologProp combined abstract syntax.**

## 5. CONCLUSIONS AND FUTURE WORK

Concrete syntax and higher-order transformations are two complementary technologies to develop grammar-based software transformation tools. Their combination simplifies the construction of such tools, but this requires some care in merging the different meta-meta, meta, and object languages involved. Surprisingly, once the mixed ASTs are built properly, the implementation of the approach in Stratego required only little and very localized changes to the existing assimilation algorithm.

The approach currently requires us to manually define the nested embeddings (cf. Figures 5 and 8); however, the nested embeddings are very similar to the simple top-level embeddings, and the necessary SDF definitions, including the two-level quotation operators could in principle be derived automatically.

So far, we have used our approach primarily with the Stratego-Stratego-WebDSL combination, in order to extend the WebDSL code generator indirectly, by systematically changing the generated code. We believe that the approach we presented in this paper can be used to implement a wide range of generator extensions and changes. Similar problems can be solved by attribute grammar forwarding [15], and while our approach works with any code generator that provides external access to the internal representation, independently of the technology used to implement it, both approaches should be compared in more detail.

**Acknowledgments.** This research was supported by EPSRC grant number EP/F052669/1.

## 6. REFERENCES

[1] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[2] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier.

```
MetaExplode =
  alltd(?ToTerm(<trm-explode>))

trm-explode =
  TrmMetaVar
  <+ TrmFromTerm
  <+ !NoAnnoList(<
        ToTermPatchFromTrm
        <+ TrmOp
      >)

MetaExplode2 =
  alltd(
    ?ToTerm(<trm-explode2>)
    + NestedFromTerm
  )

trm-explode2 =
  TrmFromTerm2
  <+ TrmMetaVar2
  <+ !NoAnnoList(<TrmOp2>)

NestedFromTerm =
  ?FromTerm(fs) ; !Op("FromTerm", <map(NestedFromTerm)>fs)
  <+ TrmOp2

ToTermPatchFromTrm : ToTerm(ts) -> Op("ToTerm", <map(trm-explode2)> [ts])

TrmOp  : op#(ts) -> Op(op, <map(trm-explode)> ts)
TrmOp2 : op#(ts) -> Op(op, <map(trm-explode2)> ts)

TrmMetaVar : meta-var(x) -> Var(x)
TrmMetaVar2 : meta-var(x) -> NoAnnoList(Op("meta-var", [<trm-explode2> x]))

TrmFromTerm  = ?FromTerm(<MetaExplode>)
TrmFromTerm2 = ?FromTerm(<MetaExplode2>)
```

**Figure 10: Generic assimilation of nested concrete syntax.**

Compiling language definitions: the ASF+SDF compiler. *TOPLAS*, 24(4): 334-368, 2002.

[3] M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. *OOPSLA-19*, pp. 365–383, 2004.

[4] M. Bravenboer and E. Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. Models in Software Engineering: Workshops and Symposia at MoDELS 2007, pp. 34–46. Springer-Verlag.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas and E. Visser, Stratego/XT 0.17. A Language and Toolset for Program Transformation, *Science Comp. Prog.*, 72(1-2):52–70, 2008.

[6] J. R. Cordy. The TXL source transformation language. *Science Comp. Prog.*, 61(3):190–210, 2006.

[7] J. R. Cordy. Eating our own dog food: DSLs for generative and transformational engineering. *GPCE'09*, *SIGPLAN Not.*, 45(2):3–4. ACM, 2009.

[8] B. Fischer and E. Visser. Retrofitting the AutoBayes Program Synthesis System with Concrete Syntax. *Domain-specific program generation*, *LNCS 3016*, pp. 239–253. Springer, 2004.

[9] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. *ECMDA-FA*, *LNCS 5562*, pp. 18–33. Springer, 2009.

[10] E. Visser. *Syntax definition for language prototyping*. PhD Thesis, University of Amsterdam, 1997.

[11] E. Visser. Meta-Programming with Concrete Object Syntax. *GPCE'02*, *LNCS 2487*, pp. 299–315. Springer, 2002.

[12] Z. Hemel, L. C. L. Kats, and E. Visser. Code Generation by Model Transformation. A Case Study in Transformation Modularity. *ICMT'08*, *LNCS 5063*, pp. 183–198. Springer, 2008.

[13] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. *GTTSE'07*, *LNCS 5235*, pp. 291–373. Springer, 2008.

[14] D. Groenewegen, *Declarative access control for WebDSL*. MSc Thesis, Delft University of Technology, 2008.

[15] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. *CC'02*, *LNCS 2304*, pp. 128–142. Springer, 2002.