Implementation of the Java-Swul language

a domain-specific language for the SWING API embedded in Java

René de Groot
rcgroot@cs.uu.nl

20th January 2005

Abstract

Java-Swul provides a concrete syntax for building a SWING user interface in Java. It is a language extension like made by a macro systems, provides domain abstractions as can be done in a functional language and is a user interface language. Java-Swul has a grammar made in SDF and uses the modularity of SDF to make a clean embedding in Java. The transformation to assimilate Swul into Java is done by a composition of rewrite rules written in Stratego. Using context information and dynamic rules the host source is also altered to allow for more complex features. We will discuss how the grammar and transformation affected the implementation efforts. And some thoughts on how to improve embeddings are also made.

Contents

Contents 3

1 Introduction 5
- 1.1 Why use Java-Swul? 5
- 1.2 How is the language used 6
- 1.3 The power of the Meta-Borg method, SDF and StrategoXT 7

2 Comparison with other methods 9
- 2.1 Language macros 9
- 2.2 Embedding a DSL in a functional language 9
- 2.3 Graphical user interface designers 10
- 2.4 Separate languages 10

3 Language design 11
- 3.1 Swul grammar 11
- 3.2 Embedding Swul in Java 11

4 The transformation 13
- 4.1 Abstract syntax tree traversal 13
- 4.2 Transformation rules 13
  - 4.2.1 Component assimilation 14
  - 4.2.2 Component properties assimilation 15
  - 4.2.3 Composition of properties and component rules 15
- 4.3 Host language contextual information 16
  - 4.3.1 Field declarations 16
  - 4.3.2 Button groups 16
  - 4.3.3 Generating event handling 17
  - 4.3.4 Static and non-static 17
- 4.4 Using the gridbag layout 18
- 4.5 Hygiene 19

5 Lessons learned 21
- 5.1 Grammar design 21
- 5.2 Transformation composition 21
- 5.3 Host language specific helpers 21

6 Future work 23

6.1 Typing and compiler integration 23
6.1.1 Type checking in Swul 23
6.1.2 Getting type information 23
6.1.3 Sharing type information 23
6.1.4 Combining multiple DSEL's in a single source file 24
6.2 Finding a place for the embedding among the developers tools 24
6.3 Abstracting over the transformation 24

7 Conclusion **25**

Bibliography **27**

Chapter 1

Introduction

Java-Swul is a domain-specific embedded language (DSEL) for creating Java SWING user interfaces. The Swul part of the name comes from **SW**ING **u**serinterface **l**anguage and embedded in Java makes Java-Swul. It was first concieved when developing the Meta-Borg [5] method. Java-Swul provides a concrete syntax for creating SWING objects. This is accomplished by writing a grammar for Swul and connecting it with the Java grammar and providing a transformation to assimilate the Swul code into normal Java code. This way the resulting source is normal Java.

The implementation and embedding of this domain specific language (DSL) is discussed in this paper. We will discuss the use of the Java-Swul language and compare it with other kinds of user interface languages. After which we will discuss how the language is designed and embedded. Furthermore many aspects of the transformation of Swul into Java will be shown. At the end we will discuss our findings during the implementation and some possibilities of further work.

1.1 Why use Java-Swul?

Building a graphical user interface in Java consists of using the SWING API in a sequence of statements. Visual components are first declared, secondly initialised and thirdly have some number of properties altered. Steps 1 and 2 can be combined, some of the work in step 3 van be done in step 2 and the actual code that does these steps can be spread around the source-files. To create a complete user interface entails combining lots of these components in a hierarchy. This hierarchy follows from the SWING API, but is heavily obfuscated in the source. When reasoning and altering the user interface the programmer reasons using this hierarchy, but can not easily convey this to the source.

Swul is a language that follows the hierarchical structure of composition used in the building of a user interface. When specifying a component all aspects of this component are concentrated in a single place. A component that provides layout for other components has these other components specified in the body of its own specification. This allows for a more natural way of transferring ideas to code. [6]

1.2 How is the language used

As stated Swul is an embedded language, so it is used inside a Java source file. Swul can be written in any place where in Java you would write an expression. In figure 1.1 the Swul is written in the right hand side of a declaration/assignment. It is interesting to compare how the same is accomplished in plain Java, as show in figure 1.2.

```
JFrame frame = frame {
  title = "Welcome!"
  content = panel of border layout {
    center = label { text = "Hello␣World" }
    south = panel of grid layout {
      row = {
        button { text = "cancel" }
        button { text = "ok" }
      }
    }
  }
};
```

Figure 1.1: A sample of Java-Swul code

```
JFrame frame = new JFrame("Welcome!");
JPanel mainPanel = new JPanel();
BorderLayout borderLayout = new BorderLayout();
JLabel helloLabel = new JLabel("Hello␣World");
JPanel southPanel = new JPanel();
JButton cancelButton = new JButton("cancel");
JButton okButton = new JButton("ok");

southPanel.setLayout(new GridLayout(1, 2) );
southPanel.add(cancelButton);
southPanel.add(okButton);

mainPanel.setLayout(borderLayout);
mainPanel.add(southPanel, BorderLayout.SOUTH);
mainPanel.add(helloLabel, BorderLayout.CENTER);

frame.setContentPane(mainPanel);
```

Figure 1.2: A sample of Java code similar to Figure 1.1

In figure 1.1 a frame is created with two properties. The value of the title is set and the content is specified. The content is a panel with a border layout which has as properties the content of the regions filled in. The composition of the user interface components can be deduced directly from the hierarchical structure of the Swul-Code.

The source in figure 1.2 accomplishes the same as the Java-Swul code. This code is also structured. The first block contains the declaration and instantiation of the components. Some properties are also set in this part since some components have parameterized constructors. In this case only the panels need more properties set then the constructors can handle. So the second and third block specify the needed layout and connects the components into the proper place in the panels.

When using the Java-Swul during your programming you would create a Foo.javaswul file. In that file you program a Foo-class which has some Swul code embedded. This Java-Swul code needs to be assimilated into Java by a preprocessor. So building the Foo class means first running `swulc -i Foo.javaswul -o Foo.java` and then further compilation can be done on the resulting `Foo.java` file.

The Swul language has many more components, more possibilities and more integrations with the Java language but this example gives a nice first glance. More examples and the current state of Java-Swul can be found at http://www.stratego-language.org/Stratego/Java-Swul.

1.3 The power of the Meta-Borg method, SDF and StrategoXT

Meta-Borg is a technology for allowing a host language to incorporate and assimilate external domains to strengthen itself [5]. This technology is used to create the Java-Swul language.

SDF [7] is a grammar formalism in which a grammar for Java-Swul is defined. SDF allows for a modular approach to grammars. A module is defined in which the Java-grammar and the Swul-grammar are loaded and production rules are defined which connect Java-nonterminals and Swul-nonterminals. An extract of this is shown in figure 3.2. These grammars are used to create parse-tables for a scannerless generalised left-right parser, in this case SGLR [15]. So now we can parse Java with Swul embedded.

The next step is to create a transformation that assimilates the Swul part of the abstract syntax tree (AST) into Java, so the entire AST is Java. This is done with StrategoXT [16]. Stratego is a language for program transformation based on the paradigm of rewriting strategies. In Stratego we define a collection of rewrite rules and combine these with strategies on where and when to apply these. After which we can generate AST's which have all the Swul assimilated into Java code. Using the pretty printer which is in the larger StrategoXT package we produce Java code from the AST.

Chapter 2

Comparison with other methods

2.1 Language macros

Macro languages are useful tools for language expansion. Macro languages come in many shapes and sizes. For all the declarative power macros can offer they are limited by the constrain that they are confined to the abstract syntax tree of the host language where they are applied. Even with a grammar defined macro system [2] all the non-terminals must map to host language non-terminals. Macro's are an abstraction over language constructs.

In [2] the transformation mapping meta-nonterminals into some host language-nonterminals is called morphing. Where assimilation overpowers morphing is that this transformation is directed by a strategy walking the entire AST of the host program, thus allowing the collection of context information and the production of effects in entirely different parts of the program. This feature is used in 4.3.3.

The aesthetics of having macros with grammar definitions, transformations-rules and pretty printing information all at once can be questioned. Assimilation as with Java-Swul offers more power, both in grammar and transformation senses. It offers a clean separation between grammar, transformation and pretty printing and a separation between making the extension and the use of the extension.

2.2 Embedding a DSL in a functional language

Functional languages with higher-order functions and powerful typing systems offer many capabilities to create a domain specific language. [8] The abstraction offered by functional composition, overloading and monads are tools to create an abstraction that serve as an DSEL.

An embedding as described in [10] for dynamically generating SQL queries, leverages the power of the type system to create syntacticly correct programs. Programmers can thus stick to a single language and retrieve the domain specific extensions from library imports. This can also be a drawback since not the same freedom as with Meta-Borg can be obtained. The domain abstraction needs to stick with style of functional programming which might not match the domain. SQL queries do not follow the syntax of the SQL language, but are constructed in a `do` notation available in the Haskell language.

The library wxHaskell [9] provides an interface to the wxWidgets GUI library. The *WX* library part of the implementation provides functional abstractions to the user interface components. This allows Haskell programs to create user interfaces and allow further programming logic to easily connect to user interface components and events. wxHaskell bears some similarities to Swul in that properties of components are set as a list of parameters after the component. Mostly the hierarchical structure of user interface components is lost in the functional abstracts.

2.3 Graphical user interface designers

A common approach to designing a graphical user interface is to use a graphical designer. These provide an easy way of clicking a user interface into existence. The designer immediately gets a good idea about how this interface will look like. The drawback is that a programmer is limited by the given possibilities of the design program. If he edits or adds to the generated Java-code this might be ignored or removed by the design program. A more common effect is that the changes are beyond the capabilities of the design program and that it will not provide any more services. In the Swul language it is possible to use Java expressions among the Swul expressions. So you can use third party visual components or your own extensions to common SWING components.

2.4 Separate languages

XUL[12] is XML based user interface language developed by the Mozilla project. It offers a way to create cross platform interfaces.

Where Java-Swul follows the Java style of layout using the layout managers, XUL has its own methods of layout based on boxes, grids and pop-ups. This enables XUL to be cross platform but fails to utilise the full capabilities of the Java platform.

Connection with external libraries is possible with XPCOM and XPConnect. XPCOM are components written in C, C++, Java Script or Python. They are connected using XPConnect, Java Scripts to interface with the XUL. With Java-Swul you can directly connect with the programming logic as you see fit. This is because Swul components are useable as Java expressions and can be bound to Java identifiers.

XUL applications can customised by using the eXtensible Binding Language (XBL). Allowing one to create his own user interface components for use in XUL. In Java-Swul all the SWING components are available and custom components can be used by falling back to Java expressions. No new language is needed to support customisation.

Other stand-alone user interface languages offer different advantages and share the same disadvantages. UIML [13] for instance provides a more cross-appliance approach to user interfaces. XForms [14] is more geared towards online forms. Just like Java-Swul XForms is not a stand-alone language. To make use of XForms one needs to embed these as part of an other markup-language. Both these languages use XML-markup to capture all the customisation of user interfaces, using XML as meta-language to describe their own language.

Chapter 3

Language design

3.1 Swul grammar

The grammar of Swul reflects the hierarchical structure of the construction of Java SWING user interfaces. Components are made by writing down a `ComponentType` which can be one of the many keywords available in Swul and specifying the properties of that component by a list of property = value assignments. The value in such an assignment is again a component. As can be seen in the SDF code in figure 3.1. A more extended version is used in the Java-Swul implementation to allow for some abbreviations, multiple component values and declaring what identifiers to use for components.

```
context-free syntax

  ComponentType "{" ComponentProp* "}" -> Component {cons("Component")}

  ComponentPropType "=" Component
                    -> ComponentProp {cons("ComponentPropSingleValue")}
```

Figure 3.1: A slimmed down version of the Swul grammar

3.2 Embedding Swul in Java

Swul is embedded in a seamless way into Java. Swul can be used in any place where you would write an expression in Java. This is accomplished by the SDF code in figure 3.2. A very small Swul expression can be ambiguous, for example a Java identifier used as expression that matches the name of a Swul component. The SGLR parser is instructed by the `avoid` parameter to solve this ambiguity. A Swul component is only useful when it is larger then a single identifier and then there will not be any ambiguity and the parser applies the `ToExpr` construction.

If it would be unavoidable or desirable to have a ambiguous embedding we could always introduce some sort of delimiters in the embedding syntax rules to solve the ambiguity.

To increase the power of Java-Swul we allow the use of Java as Swul components. When we have a property that has as value a colour, int, string, bool etc. we allow those to be

```
module Java-Swul
imports Java-15-Prefixed Swul-Prefixed
exports
  context-free syntax
    SwulComponent -> JavaExpr {avoid, cons("ToExpr")}
    JavaExpr      -> SwulComponent {avoid, cons("FromExpr")}
    JavaBlock     -> SwulComponent {avoid, cons("FromBlock")}
```

Figure 3.2: A sample of the grammar connecting Swul to Java and Java to Swul

made with a Java expression. By borrowing this from the host language we remove the need to add these in the Swul grammar and make transformations to Java for all their values.

So on places where a Swul component is expected it is possible to write a Java expression. To specify the response to an event is we even allow for an entire block of statements to be written as as right hand side.

These described grammar rules add some demands to the transformation that is needed. A component can be used as an expression, so an assimilated component should be an expression to keep the AST correct. The value of the expression should be set correctly as declared by the properties of the component before the expression is used. This leads to an identifier to fill the role of a Swul component in Java and preceding statements to fill the role of Swul properties in Java.

Chapter 4

The transformation

4.1 Abstract syntax tree traversal

Since Java-Swul can be freely used within Java, the context in which the Swul appears can differ a lot. So the assimilation of the Swul code is applied to the entire AST of the Java-Swul source file. The assimilation traverses the AST topdown. This traversal is shown in figure 4.1. Note that '<+' is a deterministic choice operator where the left hand strategy is preferred. A class declaration, class initialiser or method store are strategies that alter the context and resume the assimilation traversal. When a `ToExpr` [1] term is encountered the Swul-expression strategy matches and should succeed. The `all(s)` is a traversal operator applying the strategy `s` to all the children of the current node. Thus providing the topdown way of walking the AST.

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

Figure 4.1: The general strategy to traverse the Java AST

To hold the contextual information a number of dynamic rules[4] are created. Which are explained further in section 4.3. In section 4.2 we show how the transformation rules are applied and composed.

4.2 Transformation rules

The assimilation of the Swul code into Java code is done by a composition of transformation rules. The brunt of the work is done by simple '`a -> b where c`' rules. Pattern `a` is

[1]The production rule in figure 3.2 produces this non-terminal

matched and produces some bindings, the strategy c is applied to produce some additional rewrites and bindings and b is then built as result. These rules handle most of the component and component properties transformations. Since rules are special forms of strategies in Stratego, we use strategy operators and other strategies to combine and call the transformation rules. When we are faced with transformations that are not so straightforward as with rules then we use strategies to get the job done.

4.2.1 Component assimilation

In figure 4.2 we show an assimilation rule. The rule is named `SwulAs-JPanel`. On the left hand side there is concrete syntax [17] of a Swul component and on the right hand side concrete syntax for a Java expression. For Swul syntax embedded in Stratego the identifier `ps*` is a special escape identifier and provides a binding to a list of component properties. `x` is an annotation to the Swul term that holds the identifier for which a type declaration has been made at the proper place. This can either be a local variable or a global variable. (See section 4.3.1) for more details.)

```
SwulAs-JPanel :
  swul c |[ panel { ps* } ]|{x}
  ->
  expr |[ {| x = new JPanel() ; bstm* | x  |} ]|
    where <map(SwulAs-JPanelProp(|x))> ps* => bstm*
```

Figure 4.2: The transformation rule to assimilate a panel component

In the `where` clause the list of component properties is assimilated to a list of statements. This is done by the `map(s)` strategy that maps the strategy (or rule) `s` to each element in the list. The identifier of the component is passed along to the rule so the generated statements are applied to the proper variable. The resulting Java code is an instantiation of the Swing component followed by the statements that alter Swing component. Since an expression is required as result of the transformation `x` representing the Swing component identifier added.

Making it possible that a Swul component needs to be transformed into a Java expression makes incorporating Java into Swul easy. At the place where one could write a Swul component it is also possible to write a Java expression. This is done by the `FromExpr` grammar rule in figure 3.2. The transformation rule for components has as alternative that matches the the `FromExpr` term and builds the expression inside as the result of the assimilation.

In the right hand side where the Java expression is built some special delimiters occur. The Java here is written in EBlock form (further described in section 5.3). This is a form provided by the java-front [5] package. That same package provides a transformation filter to rewrite EBlocks to normal Java. The escape with `{| statements | expression |}` is an expression that needs the statements lifted to precede it. The strategy core-lift-eblocks lifts these statements in front of the expression, leaving the expression behind.

4.2.2 Component properties assimilation

The properties of components have transformation rules that are similarly constructed as the rules for components. The parameter x is the identifier of the Swing component. If this Swing component has a layout then y is the identifier of that layout. No annotation is added to the term since a property will not produce an expression but a statement.

```
SwulAs-BorderLayoutProp(|x,y) :
  swul cp |[ cpt = c ]| -> bstm |[ x.add( e1, e2 ) ; ]|
    where <SwulAs-Component> c        => e1
        ; <SwulAs-LayoutRegion> cpt   => e2
```

Figure 4.3: How a region of a border layout is assimilated

Since `SwulAs-BorderLayoutProp(|x,y)` is called by the `border layout` component rule we know we are faced with a property of the border layout. For this rule to succeed both the rules in the where clause must succeed. So the statement is only built when the property is a correct border layout property.

4.2.3 Composition of properties and component rules

As we have described component property assimilation rules are chosen with regard to which component is assimilated. A panel component assimilation rule uses panel properties rules to create statements. A panel can contain other components, because it is a subclass of `java.awt.Container`. The assimilation rules of a container should be used for this. We therefore make sure that the property of a component type A can also be a property of a component type B if A maps to a superclass of B. A small example of this is shown in figure 4.4. Note that in Stratego a rule name may be used for multiple rules, these are combined to a single rule by an implicit non-deterministic choice operator.

```
SwulAs-JPanelProp(|x)     = SwulAs-JComponentProp(|x)

SwulAs-JComponentProp(|x) = SwulAs-ContainerProp(|x)

SwulAs-ContainerProp(|x)  = SwulAs-ComponentProp(|x)
SwulAs-ContainerProp(|x) :
  swul cp |[ layout = c ]| -> ...
```

Figure 4.4: Mapping method inheritance to property rule composition

In the right hand side of a border layout property we expect some component to be declared. This component needs to be mapped to some expression that is of the type `java.awt.Component` or subclass of that type for the add method to be called correctly. So we apply the `SwulAs-Component` rule (Figure 4.5). This rule is a composition of all the components that map to a subclass of `Component`. In Swul there is no 'Swing component' component so this composition makes up the entire component rule.

Smaller but similar composition is done with layout managers and menu items. We only apply rules that have the right type as result. Thus if a rule succeeds the Swul was of a

```
SwulAs-Component   = SwulAs-Container

SwulAs-Container   = SwulAs-JComponent
                   + SwulAs-Window
```

Figure 4.5: Mapping class hierarchy to component rule composition

correct type and the resulting Java is also of the correct type. The type information given by the API is thus kept within the (composition of) the transformation rules.

4.3 Host language contextual information

4.3.1 Field declarations

In the context of a class a list of field declarations for components that need to be declared global is kept. These are button groups and explicitly named components.

For example the cancel-button in figure 4.6 can be named as such:
`button named cancelButton { text = "cancel" }` This will have as effect that this button will be declared global for the class with the identifier cancelButton. In this way declarations can be made without losing the hierarchical structure and components can be easily accessed, altered and used by hosting Java code.

```
JFrame frame = frame {
  title = "Welcome!"
  content = panel of border layout {
    center = new MyCanvas()
    south = panel of grid layout {
      row = {
        button { text = "cancel" }
        button {
          text = "exit"
          enabled = false
          action event = {
            System.err.println("stopping␣execution");
            System.exit(0);
          }
        }
      }
    }
  }
};
```

Figure 4.6: Java-Swul with Java-expressions and statements

4.3.2 Button groups

Some user interface components in Java can be grouped. These are for example `JToggleButtons` and `JRadioButtons`. The state of one component depends on the state of the other components in the Button Group. These components in Swul are grouped by setting the group

property. In the context of a class a mapping from groups to the identifiers which have a ButtonGroup assigned is kept in a dynamic rule. This dynamic rule is used to produce the statement required to add Swing components to a group. The field-declaration of a Button Group is generated when an unknown group identifier is first seen.

4.3.3 Generating event handling

The Sun documentation on events offer two different ways of implementing event handling in Java. First and most common is to create classes that implement EventListener. A clean implementation of this often needs quite a few extra (inner-)classes. More classes mean a bigger memory footprint and large dispersion of code with lots of layout. Anonymous inner classes allow one to centralise the code but this makes code difficult to read.

The Event Handler class in the Java API can efficiently handle events, but is less powerful. Event Handler uses Proxy-classes to implement the different listeners. A created Event Handler responds to events with a call to a method. This method is passed to the Event Handler by representing it as a string. The parameter to be used when calling this method is specified with a string. This string represents a method call to acquire a value for the parameter. This makes using the Event Handler very error-prone and hard to debug.

```
menubar = {
  menu { text = "File"
         items = {
           menu item { text = "New"
                       action event = { controller.createNewDocument(); }
           }
           menu item { text = "Open"
                       action event = { controller.openNewDocument(); }
} } }      }
```

Figure 4.7: Creation of a menubar with a 'File' menu with 2 items and actions

In Java-Swul events can handled by declaring this as a property of an component. The event type is set with the response, which is a block of Java code (see Figure 4.7). This block of code gets placed in a method of an inner class and an Event Handler is created to connect the two (see Figure 4.8). When the inner class is created, it is taken into account if the context is static or not.

The code blocks which are placed into an inner class have equal expression power as a self made inner classes, but fewer inner classes are needed. In this example the footprint of the program is just as large as with two separate inner classes. One object for the Event Handler and one for the generated inner class. Every next action event that is handled will not add to the size of the footprint. The Event Handler uses a proxy for each Event Listener and the handling inner class is just enlarged. So we have the ease of using inner classes, the concentration of code as with anonymous inner classes and the efficiency of Event Handlers.

4.3.4 Static and non-static

The modifiers to be applied to inner classes and field declarations are the result of a static/non-static context change. The modifiers starts out with Private and might be-

```
class Foo {
  private InstanceHandler_0 instanceHandler_0 = new InstanceHandler_0();

  public void Bar() {
    jMenuItem_0.addActionListener((ActionListener)
      EventHandler.create(ActionListener.class, instanceHandler_0,
                          "actionListener_0",    ""));
  }

  public class InstanceHandler_0 {
    public void actionListener_0(ActionEvent event) {
      controller.createNewDocument();
    }
  }
}
```

Figure 4.8: Excerpts of the generated eventhandling code

comes `Private Static` when a static context is entered. Entering a static context is noticed by the strategies in `swul-assimilate` (See figure 4.1).

4.4 Using the gridbag layout

The gridbad layout is a good example of where the API is really distant from the actual concepts. Swing components are placed within a grid and can be multiple rows wide and/or multiple columns height. Of course the width, height and coordinates are expressed in numbers, but one does not reason about the relative placements of components in these numbers. To create a textual environment that is closer to the concepts of the layout we introduce some special components.

A gridbag layout component type has as properties rows with lists of components. Each row maps to a row in the grid. The implicit indexes in these rows map to the columns in the grid. Beside the normal components you can write `^^^`, <<< or `___`. The special component `^^^` can be written below a normal component and signifies that the above component should be stretched down. <<< has a similar meaning, but is placed right of a component and entails a horizontal stretching. The component `___` is a place filler to use in gaps in the grid. An example of this textual grid drawing is shown in figure 4.9. The end result of this gridbag layout can be seen in the screenshot show in figure 4.10.

```
center = panel of gridbag layout {
  row = { scrollpane of tree{} label { text = "Edit"}             <<< }
  row = { ^^^                  scrollpane of new JTextArea(20,20) <<< }
  row = { ^^^                  new JButton("Add") new JButton("Done") }
}
```

Figure 4.9: Code example of useing the gridbag layout in a panel

The transformation of this grid to Java statements means that some calculations need to be performed. This is done by a combination of strategies that perform a bottomup-traversal of the list of rows. The traversal is started in the lower-right corner, moves from right

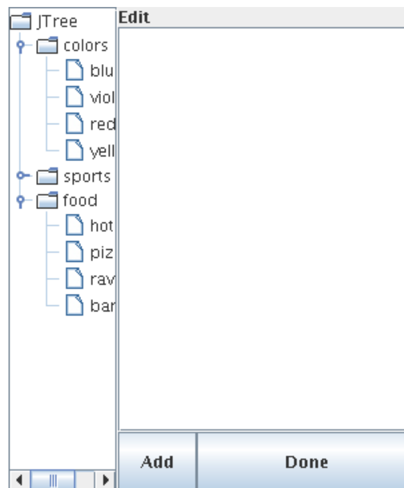

Figure 4.10: Screenshot with the result of a gridbag layout in Java-Swul

to left and visits all the rows from the last to the first. When a special grid component is encountered this alters the width or height of an unvisited cell. When a component is encountered it is created and added to the location in the layout grid with the current coordinates and collected width and height.

4.5 Hygiene

Hygiene of Java-Swul is a desirable property. Since the Swul assimilation can produce new inner-classes and new declared variables there are some dangers of shadowing variables and such. In [1] the solution for this with a preprocessor is described to be variable renaming and a detailed code walk.

In Java-Swul new names for classes and identifiers are generated using the `newname` strategy. This is part of the Stratego standard library and produces strings that are not found in any other part of the AST. This is because the complete source AST is represented by an ATerm [3] which ensures maximum sharing. The `newname` strategies uses this maximum sharing to create a complete new term. Combined with that the effects of the Swul assimilation do not reach beyond the given source, gives full hygiene for Java-Swul.

Chapter 5

Lessons learned

5.1 Grammar design

The initial implementations of Java-Swul consisted of a large grammar definition. The syntax rules were inspired by the type information in the SWING API. New features often required new non-terminals and syntax rules. This leaved less work for the transformation and allowed for more general transformation rules and a simpler rule composition.

When an error was made in the declaration of a property or component the parser would produce the error. For all the merits of the SGLR parser its error reports are not really friendly. The errors shows the exact symbol that can not be understood and that is not helpful. Overall the larger syntax and smaller transformation was not a good approach.

A general syntax definition as is currently implemented provides a smoother parsing of Java-Swul source. This makes the concrete syntax in the Stratego source also a lot easier to use. The amount of special escaped identifiers lessens and the amount of possible escapes from Stratego to Swul also decrease. The overall quality of the transformation rules also increased with the added responsibilities of the transformation.

5.2 Transformation composition

The normal way of extending a programming language with domain specific capabilities is with libraries. Many a domain has an API to provide extra reusable functionality. It is a good practise to use these API's as back end of a DSEL. Much implementation and design is found in these API's and both can thus be reused. This reduces the amount of work done in the transformation and offers a guide to designing the transformation.

In Java-Swul the typing and class structure of the SWING API is mapped to the composition of the transformation rules. (see section 4.2.3) This gives a structured way of creating and adding capabilities to the Swul language.

5.3 Host language specific helpers

In section 4.2.1 we describe the usage of transformation rules. The usage of EBlocks provides a straightforward way to keep the rules clean and readable. The rules can be

trusted to fill a place in the Java AST, such as an expression or a statement. The rules also have a way to add additional statements without leading attention away with the fitting of them into the AST.

Currently the EBlocks offer statements preceding an expression in place of an expression, an expression followed by statements as an expression and a list of statements to be placed in a single statement. These are of great help in making and composing the transformation rules for the embedding.

Chapter 6

Future work

6.1 Typing and compiler integration

6.1.1 Type checking in Swul

The Swul assimilation into Java has type information in the transformation. The applying of these rules fails if components or component properties are of an incorrect type. As a result the entire transformation fails and produces lots of errors [1]. A next logical step is to handle incorrect Swul that does not transform and produce friendly errors and pretty printing of the incorrect code.

6.1.2 Getting type information

Another interesting question is how to make sure embedded Java expressions in Swul are of a correct type. We know they are syntacticly well formed, but the type of an expression is unknown and generally can not be derived from a single expression. To find out the type of an expression help is needed from the Java compiler. The type information from the Java compiler might not be sufficient. For example, an identifier used as expression in a Swul expression may be originate from an other Swul expression or be the result of a declaration in the current Swul expression.

6.1.3 Sharing type information

There has to be some sort of interaction between the Java compiler and the Swul assimilation to share type information gathered from the Swul code. It is of course possible to type check the resulting Java source. The errors that are found then could be mapped back to the original Swul code. Although this is doable for humans, this is not very convenient and desirable. Automating this process of interpreting Java compiler errors into the Swul context is also not a wise course of action. The scope of errors of the Java compiler are wide and are intended for the human reader, not for a formal system.

A complete solution would entail a compiler that

- can use extra parse tables to parse an embedded language

[1] These errors are from the Java pretty printer encountering unknown Swul code.

- calls the assimilation of the embedded language at the proper time
- calls upon the assimilation process to provide type information of embedded constructs
- can provide type information to an assimilation process
- can receive and store type information generated in the assimilation

6.1.4 Combining multiple DSEL's in a single source file

With the implementation of the assimilation process as a preprocessor for a Java source file the preprocessor needs to parse the Java file. This left no room for an other embedding. The need to do the double work of parsing the entire Java file would not be needed if this was solely done by the compiler. A compiler that is open to embedded languages could even cope with multiple embedded languages that have no further knowledge of each other.

6.2 Finding a place for the embedding among the developers tools

The gained ease of programming in a more domain-oriented language falters when complex and integrated development environments are used. These programming environments 'understand' the code you are writing. They give tips, help with refactoring, highlight syntax and offer debugging. These tools are no longer available when the source has an unknown language embedded, not even for the surrounding host code. Although these tools still work for the generated code, the usefulness at that point is doubtful. An interesting view is shown in [6] that sees the creation of tools for a DSL as integral part of creating a DSL.

The building process also needs to be altered to apply the preprocessor to the source with embedding. This is often possible, when the user may specify or alter the build process with XML or Make-like configurations.

6.3 Abstracting over the transformation

Application libraries are the basic form of embedding [11]. Libraries provide the building blocks for a programmer. Offering a concrete syntax for using these blocks, as done with Java-Swul, could also work for other domains and libraries. It would be interesting to see if the approach of using the API design as inspiration for the assimilation rules composition would also work with other API's and their embeddings.

Chapter 7

Conclusion

Java-Swul is a useful tool for creating user interfaces. The fact that it is embedded and deeply connected with its host language adds many positives and unique features. The tools used to create this language are powerful and easily up to the task. The grammar and parser tools are also powerful and ease the embedding. The assimilation of the Swul into Java is clear and lucid. The use of concrete syntax in the transformation rules makes them easy to understand and syntacticly correct. The composition of the rules offers a good way to steer the transformation. More elaborate transformations are also possible with the use of dynamic rules and more complex strategies. Syntactic correctness and hygiene can both be taken care of with a minimum of effort.

There still lies work ahead to broaden the embedding towards the compiler and towards the programming tools. As well is there interest to gain more experience with embeddings of other domains.

Bibliography

[1] Bachrach, J., and Playford, K. The java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications* (Nov. 2001), vol. 36(11) of *SIGPLAN Notices*, pp. 31–42.

[2] Brabrand, C., and Schwartzbach, M. I. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation* (2002), ACM Press, pp. 31–40.

[3] Brandvan den Brand, M. G. J., de Jong, H., Klint, P., and Olivier, P. Efficient annotated terms. *Software, Practice & Experience 30*, 3 (2000), 259–291.

[4] Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae* (2005). (Conditionally accepted).

[5] Bravenboer, M., and Visser, E. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)* (Vancouver, Canada, October 2004), D. C. Schmidt, Ed., ACM SIGPLAN.

[6] Dmitriev, S. Language oriented programming: The next programming paradigm. http://www.onboard.jetbrains.com/articles/04/10/lop/.

[7] Heering, C., Hendriks, P. R. H., Klint, P., and Rekers, J. The syntax definition formalism SDF - reference manual -. *SIGPLAN Notices 24*, 11 (1989), 43–75.

[8] Hudak, P. Building domain-specific embedded languages. *ACM Comput. Surv. 28*, 4es (1996), 196.

[9] Leijen, D. wxhaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)* (Sept. 2004), ACM Press.

[10] Leijen, D., and Meijer, E. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)* (Austin, Texas, Oct. 1999), pp. 109–122. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[11] M. Mernik, J. H., and Sloane, A. When and how to develop domain-specific languages, 2003.

[12] HTTP://WWW.MOZILLA.ORG/PROJECTS/XUL/XUL.HTML. Xml user interface language (xul) 1.0.

[13] HTTP://WWW.UIML.ORG/INDEX.PHP. Home of the user interface markup language.

[14] HTTP://WWW.W3.ORG/TR/XFORMS/. Xforms 1.0.

[15] van den Brand, M. G. J., Scheerder, J., Vinju, J., and Visser, E. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction (CC'02)* (Grenoble, France, April 2002), N. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–158.

[16] Visser, E. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)* (May 2001), A. Middeldorp, Ed., vol. 2051 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 357–361.

[17] Visser, E. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE'02)* (Pittsburgh, PA, USA, October 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–315.