Component-based DSL Development

Thomas Cleenewerck

Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
thomas.cleenewerck@vub.ac.be
http://prog.vub.ac.be/

Abstract. Domain specific languages (DSLs) have proven to be a very adequate mechanism to encapsulate and hide the complex implementation details of component-based software development. Since evolution lies at the heart of any software system the DSLs that were built around them must evolve as well. In this paper we identify important issues that cause a DSL implementation to be very rigid in which all phases are tightly coupled and highly dependent upon one another. To increase the poor evolvability of current day DSL development environments a new development environment Keyword based programming (KBP) is proposed where DSLs are built by using a language specification to compose and glue loosely coupled and independent language components (called keywords).

1 Introduction

Component-based software development has proven to be a significant improvement over traditional software development methods, and is well on its way to become the dominating software development paradigm. Nowadays, many software systems are assembled out of reusable and stand-alone parts. However, such composition of components is far from an easy task. It requires selecting the most appropriate components, solving data format and architectural mismatches, adapting to the application context, etc [SN99,LB00]. To encapsulate and hide these complex implementation details, domain specific languages (DSL) have been proposed [Sin96,SB97]. Unlike general purpose languages, DSLs are little languages that are expressive over a particular domain. Using DSLs, users can write higher-level, domain-specific descriptions (DSD) or programs using domain terminology. The code generator that produces an executable program out of the DSD contains the actual implementation details, e.g. the most appropriate components, their interface, the composition rules, the code to glue and adapt components, etc.

Unfortunately, developing DSLs is hard and costly. Therefore, their development is only feasible for mature enough domains. Such domains have been extensively analysed, knowledge about it is considered stable, and their components have been proven to be truly reusable and composable. As such, it is believed the DSL can be developed and will remain stable. Examples of those well-established domains are image manipulation, text processing, database interaction, etc. However, it takes a lot of time before a domain is considered stable (it took us more than a decade to reach a relative stable GUI building library [GS94]). And, even then we cannot avoid changes, since it is generally known that evolution lies at the heart of any software system [MM99]. As such, DSLs need to evolve, even when they are developed for mature enough domains.

Although current DSL development technologies have boosted DSL development, they only freed us from tedious tasks like lexing, parsing, pattern matching, manipulating and transforming an abstract syntax tree (AST). At this low level, a developer cannot help but implement a rigid DSL in which all phases are tightly coupled and highly dependent upon one another. As a consequence, the DSL becomes hard to evolve. To alleviate this problem, we need an advanced DSL development environment, that takes evolution into account and allows us to decouple the different phases and remove unnecessary dependencies. The purpose of this paper is therefore two-fold. First, we identify some important issues related to evolution in current-day DSL development environment. Second, we propose adequate solutions to these problems, that allow us to define DSLs that can be evolved in a more easy and straightforward way.

The paper is structured as follows. Section 2 explains the problem statement in more detail. Section 3 introduces the solution we propose to tackle the problems. Section 4 reflects on the proposed approach. Section 5 discusses related work, while section 6 presents our conclusions.

2 Problem Statement

In this section, we will identify the major problems developers are faced with when evolving a DSL. First, we will discuss the general architecture of a DSL development environment. Next, we introduce the running example, that we will use and gradually evolve throughout the paper to illustrate the problems.

2.1 General DSL Development Environment Architecture

In the general development environment architecture for DSLs consists of tree parts:

1. the grammar of the language, specifying the syntax and the set of correct sentences;
2. the library of components described in the target language;
3. the transformations that define the DSL's semantics

The grammar is usually defined using Backus-Naur-Form (BNF) [NAU60] or some of its variants. A parser processes a character stream according to the grammar and produces an abstract syntax tree (AST), representing the parse tree of a program written in the DSL. The transformations transform this AST to the AST of a program written in the target language, that uses the library of components. Usually the target language is a general purpose programming language (GPL), but it can also be a another DSL.

2.2 Running example

Throughout this paper we will use a single simple example. In this example, a GUI library, written in Visual Basic, will be encapsulated with a DSL. Initially the GUI library contains only one component: a label. Because labels are widgets with a rather complex interface, we want to encapsulate the common usages of this interface in an easier-to-use DSL language. For the moment, the only expression/sentence that we can write in the DSL is the following:

```
label "title"
```

This program defines a label with caption `title`. As can be seen, a developer doesn't need to know which component he has to use, nor how it should be used, how it should be instantiated or how the caption should be set.

To implement this DSL, we need to specify its syntax and the transformations that translate the expression `label "title"` into code in some executable target language. The syntax is specified in the grammar which is often written in some BNF variant. The following two BNF-rules specify the syntax of the 'label' expression:

```
label        ::= "label" labelcaption
labelcaption ::= string
```

They state that the expression must start with the word '`label`', followed by a *labelcaption* sentence. The latter is defined in the second rule, which states that a labelcaption is simply a string.

Based on this grammer, the parser can generate an abstract syntax tree, which is manipulated by the transformations associated with the DSL. We will specify these transformations in pseudo syntax, that is close to existing transformation systems such as ASF+SDF and Khepera [vdBK02,FNP], to abstract away from irrelevant technical details.

Below is the transformation that translated the DSD expression `label "title"` to the target language. It will transform the expression into the equivalent two Visual Basic expression (lines 5 and 8):

```
(1)  transform label
(2)  define
(3)    A arg(1)
(4)  by VBLabel
(5)    Begin VB.Label
(6)        name = "mylabel"
(7)    End
(8)  by VBSub
(9)    Public Sub Form_Load()
(10)        mylabel.caption = %A%
(11)   End Sub
```

Conceptually, the transformation consists of 3 sections:

- The first section defines the node of the AST that can be transformed by the transformation. In this case, line 1 states that all label nodes of the AST tree will be transformed;
- The second section defines the variables that can be used in the target language structure to retrieve information out of the AST node. For example, line 3 defines the variable A and initializes it with the first element of the label node, i.e the string denoting the labelcaption.
- The last section defines the target language structure that will replace the matched node in the AST. In this example, lines 4 and 8 state that the structure of the target language which is generated is respectively a VB.Label and a VB.Form. Lines 5 to 7 and 9 to 11 generate the resulting expression written in the syntax of the target language.

In the following subsections the GUI library will be gradually extended with new features and components. Naturally the DSL will have to keep up and evolve as well. We will explain the problems we encounter.

2.3 Information Exchange

A transformation defines which elements of an AST tree it transforms. The `label` transformation from the previous section, for example, transforms the label node of the source AST. It uses the label caption (stored in variable A) to produce a VB.Label node of the target AST. The caption is stored in the label source AST node and is locally accessible. We call such information *local information.*

When developing more complex DSLs, transformations often need to access information that is not available locally, but is available elsewhere in the AST. Such information is called *non-local information.*

In what follows, we will gradually extend our running example. First, we will show that the evolvability of a DSL is hampered if information is hard-coded into the transformations. Second, we will argue that simply parameterizing the transformation, and passing the information in some way or other, does not resolve the evolvability problem.

Adding Support for Multilinguality, First Try Suppose we want to extend our GUI library with support for multilinguality. This requires us to add a *translator* component, that translates strings to another language. Naturally, we need to adapt the DSL accordingly, so that it takes this new component into account. In a first stage, the translation will be triggered automatically, so the DSL's syntax is left unchanged: The DSL program itself thus also remains the same:

```
label "title"
```

Clearly, this example once again illustrates that DSLs abstract away from implementation details, which allows the component library to change without affecting DSL programs.

The translation extension requires the introduction of a new transformation, which is implemented as follows:

```
(1)  transform string in labelcaption
(2)  define
(3)    A arg(1)
(4)  by VBExpression
(5)    translator.translate(%A%, "dutch")
```

This transformation will be triggered for every string node inside a labelcaption node (line 1). The string value itself is stored in variable A (line 3). The result of the transformation is an expression (line 5) that calls the `translate` method on a translator component. This method takes two arguments: the string (caption of the label) that needs to be translated and the language of user interface.

When we apply these transformations to our domain-specific program, the following code is generated:

```
Begin VB.Label
     name = "mylabel"
End
Public Sub Form_Load()
        mylabel.caption =  translator.translate("title", "dutch")
End Sub
```

Although this code seems correct, we still identify two major problems with the definition of the above transformation. First of all, the language of the user interface is hard coded in the transformation, and can thus only be changed by changing the transformation. This seriously hampers its evolvability. Second, the translator component that is accessed through the variable `translator`, used in the resulting program. Clearly, this variable should be defined elsewhere in the program, but should be accessible from within the subroutine, according to the scoping rules of the Visual Basic programming language. The translate transformation cannot guarantee this however, since it does not know which code the other transformations generate.

Based on these two observations, we conclude that the evolvability of the DSL is constrained in two ways: the UI language needs to be defined outside the transformation, as is the expression used to access the translator component.

Adding Support for Multiple Languages, Second Try To avoid hard-coding the UI language into the transformation, we can allow the developer to specify the language elsewhere in the DSD. For example the UI language could be retrieved from a configuration file. The file has a standard property bag containing a list of key-value pairs. Such change requires us to extend the DSL with three new BNF rules. The first states that the 'language' expression must start with the `language` literal followed by two strings denoting the file and the property in the file, respectively. The file is a string (second rule) and the property is an identifier (third rule).

```
language ::= "language" file property
file     ::= string
property ::= id.
```

Our example DSL program now becomes:

```
language "config.ini" UILanguage
```

The translator transformation must now retrieve the UI language to be used out of the AST tree of the DSL program. We will refer to this kind of information as *non-local information* because it is located in another node of the AST tree than the node on which the transformation is applied. The code bellow illustrates the approach taken. In this transformation two new variables are defined (lines 3-4) where the variable C is initialized with the property denoting the UI language in the configuration file.

```
(1)  transform string in labelcaption
(2)  define
(3)      A arg(1)
(4)      C (getParent().getChild(language).getChild(Property))
(6)  by VBExpression
(7)     translator.translate(%A%, configfile.getProperty(%C%))
```

The code or expression that locates and retrieves information somewhere in the AST tree is still a stain on the evolvability of the DSL. Such code or such expression depends in most cases heavily of the particular structure of the AST tree. In the translate transformation the configuration file property containing the UI Language is retrieved by the expression on line (7). The expression contains very detailed information about the location of the information and how to traverse the AST to get to this location. Suppose we extend the GUI library with components to make the UI language configurable within the application itself. Because of the addition of new BNF rules and transformations to support this extension in the DSL, the UI language is contained in other AST nodes. Consequently the expression to locate and retrieve non-local information must changed. So whenever the DSL grammar changes and thus the AST tree, the whole set of transformations must be examined and checked to determine if they are affected and possible invalided because of the changes to the DSL.

Transformations are thus tightly coupled with the overall language structure, requiring permanent maintenance and therefore limiting the evolvability of the domain language.

2.4 Composition of Transformations

In the previous section, we have shown how the fetching of non-local information can tightly couple a transformation to the language specification. We will now show how the *scope of the transformation* also introduces tight coupling with the language specification and prevents the arbitrary composition of these transformations. As a result, transformations cannot be reused in different contexts of the language. Once again, this complicates DSL evolution.

The scope of a transformation is the definition of the regions in the AST tree which will be transformed. Consider for example the translate transformation, it translates only strings within (read, in the context of) labelcaptions. This is specified in the first line of the transformation:

```
(1)  transform string in labelcaption
(2)  ....
```

Hence the translate transformation will only be triggered in a labelcaption. When the DSL evolves the translate transformation must be revised and possibly changed against the new grammar. This forms a major obstacle for evolving the DSL and the free composition of transformations. To illustrate this problem, consider an extension of our GUI library with forms which are containers for labels. The DSL will be extended with the following bnf rule:

```
form          :- "form" id "title" string
```

The rule states that a form starts with the form-literal followed by its name, a title-literal and a string. Since the library offers support for multilinguality, the title of the form must be translated as well. Although the DSL already contains the translate transformation, it cannot be reused for this purpose, because its scope is defined and fixed in the transformation itself, and limited to the translation of label captions. Consequently the transformation is tightly coupled to the overall language context, in this case to the part where the labelcaption is a string. To be able to reuse the transformation and let evolve the domain language more easily, the scope of a transformation should be defined *outside* the transformation.

One could argue to just remove the scoping information out of the transformation (as is given below).

```
(1)  transform string
(2)  ...
```

The problem with this transformation is now that all strings of the DSL will be translated wherever they occur in the AST tree. These kind of uncontrolled transformations are certainly not what we want because they can seriously disrupt and corrupt the transformation process.

3 Proposed Solution: Keyword Based Programming

To render the implementation of a DSL language more evolvable the transformations need to be decoupled from the overall language structure and other transformations. Low coupling and high cohesion are the key factors to enable reusable and composable transformations which lead to a more evolvable DSL implementation. To increase the reuse of transformations we need to be able to encapsulate and parameterize them with non-local and other configuration information. To increase the composability of a transformation the scope of a transformation should be defined outside the transformation itself. Let's introduce the general ideas first by showing the stripped version of the translate transformation that is parameterized by the UI language to use (parameter `C`) and how to obtain the translator service (parameter `B`). To be able to use this translate transformation you must bound the parameter `C` to the UI language and bound the parameter `B` to an expression which returns a component with the translate service and specify the scope the transformation to the different contexts in which it used, e.g. within the label and the form.

```
transform string
define
    A arg(1)
    C
    B
by VBExpression
    %B%.translate(%A%, %C%)
```

The coupling of the translate transformation with the part of the DSL language that specifies the UI language, the component to translate the strings and the definition of the strings which need to be translated have been removed. When the DSL evolves all the transformations don't need to be scanned and checked for consistency with the overall language structure. Instead only the parameters to configure the transformations need to be checked. As you can see, the transformation doesn't contain any information about the DSL language *for which it was defined.* Actually we can now rephrase the last sentence into: the transformation doesn't contain any context information *in which it is used.*

We have developed the *Keyword Based Programming (KBP)* DSL development environment that implements the ideas stated above.

3.1 Architecture

KBP is a DSL development environment (DE) where DSL are built via the composition of language components. The architecture is quite different compared to the architecture of traditional DE and most other DSL DE's that are based on the former (cfr. section 5). The architecture of traditional DE is structured according to the functional layers in the system, e.g. lexing, parsing, generating, transforming and finally code generation. Functional decomposition results in rigid models where language features are scattered across the different functional layers. In contrast, the architecture of the KBP DE (figure 1) reflects a structural decomposition. We followed the component based approach where each component contains the necessary information to recognize and parse its syntax and contains its semantics to transform itself by producing a result. These components, called language components or keywords, are composed and glued together by a language specification that provides the necessary context information to the keywords. Let us discuss the two parts: *the language specification* and the language components called *keywords* in more detail in the next two sections.

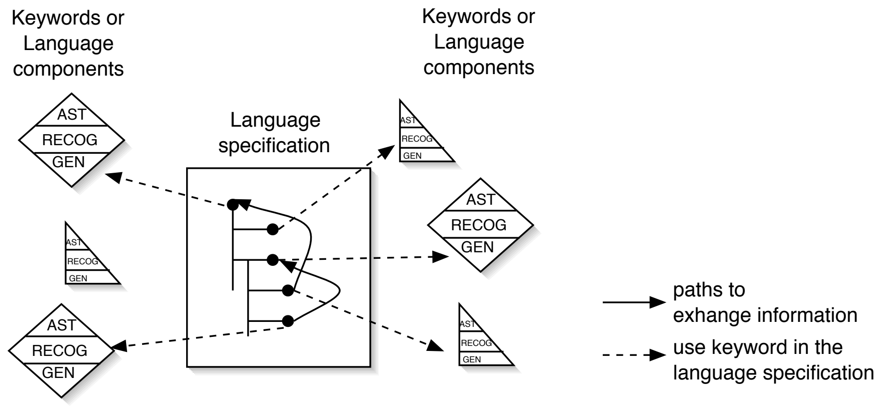
Keywords or Language components
Keywords or Language components
Language specification
AST
RECOG
GEN
paths to exhange information
use keyword in the language specification

Fig. 1. General overview of the approach. In the middle of the figure the language specification is shown. It composes, glues and configures keywords or language components. A keyword consists of three parts: a definition of its AST node (AST), a recognition pattern describing its syntax (RECOG) and a generator defining its semantics (GEN). The full arrows between the various parts of the language specification symbolizes the paths between the the keywords to exchange non-local information.

3.2 Keywords

Each transformation of the previous section is part of a single keyword in KBP. Keywords are stand-alone components comprising a single fined-grained language feature. The term keyword may be somewhat confusing, but in KBP a keyword doesn't have to start with a reserved word. Keywords can be a Java for-statement, a delimiter e.g a dot or a semi-column, a block-statement, or can have no syntax of its-own like the translator-keyword (i.e. the translator transformation introduced in the previous section will be written as a keyword). A keyword definition consist of four parts: (1) a name, (2) a recognition pattern describing the syntax of the keyword, (3) the AST node that needs to be built during parsing to contain the parsed information and (4) a generator which implements the semantics. The code below is the skeleton of a keyword where: XXX is the name of the keyword (line 1), YYY is the name of the AST node (line 2) and ZZZ (line 5) is the type of the structure that will be returned when generating the keyword.

```
(1) keyword XXX {
(2)    ast { public class YYY { ... } }
(3)    recognition pattern { ... }
(4)    generator {
(5)       public ZZZ generate() {
(6)              return ... ;
(7)       }
(8)    }
(9) }
```

The recognition pattern describes the syntax of the language component. Parsing is done according to this pattern. The pattern is described using a syntax close to regular expressions. During parsing the recognized information is stored in the AST node (the second part of a keyword). The AST node is a java class definition augmented with some new constructs to facilitate the definition, e.g. `keyword c` (which will be further explained in an example). The last part of a keyword is called the generator. The generator is a Java method containing the code that produces a new target language structure, hereby using the information stored in the AST node defined by the keyword. Apart form the java constructs, additional constructs are provided here as well, e.g. `generate c`. In KBP terminology the actual transformation is called a generator because it only produces some results rather than transformations that besides producing, also change the AST tree.

Let's revise the first part of the language that we gradually evolved in the problem statement in KBP. We will not go in all the details of the environment because in this paper we focus on the dependencies inside the transformations. The code below is the definition of the label keyword. Before we go into the details of definition, a keyword can be parameterized with other keywords through keyword parameters defined in the AST, recognition pattern and generator. The label keyword has one parameter `c` which is the caption of the label. To avoid any confusion the expression '`keyword c`' is not the definition of the '`c`' keyword, but is a formal parameter. In the language specification (which will be introduced later on) keywords can be bound to it.

Let's walk over the definition of the label-keyword step by step. The recognition pattern states that in a DSD one should write the word `label` first followed by an expression that matches the recognition pattern of the keyword in the parameter `c`. During parsing the AST node (defined right above the recognition pattern) will contain a reference to the AST node defined by the keyword in the parameter `c`. In the last part of the keyword the generator is defined. The keyword returns a object of type `VB.Label`. First the keyword in the parameter `c` is generated, which can trigger the generation of a chain of keywords. Its result is stored in the variable A. Note that the variable has type `VB.Expression`. This implies only that the *result* of the generation of the `keyword c` must be a Visual

Basic expression. Afterwards a new typed target language structure (`#VB.Label` and `#VB.Sub`) is created which will be returned as the result of the keyword. The target structures that are returned are AST nodes of the target language. To avoid the need to create and initialize them manually e.g.

```
VB.ASTLabel lab = new VB.ASTLabel();
lab.name = "mylabel"
VB.ASTSub sub = new VB.ASTSub();
sub.modifiers.add(Modifiers.PUBLIC);
sub.name="Form_Load";
sub.statements = ...
```

A new construct is added of the form `#Language.Type{ ... }`. This will create an appropriate AST node of type `Type`, and parse the content between the curly brackets. During parsing the AST node will be probably initialized. Expressions of the form '`%=XXX%`' inside the target language structure denote variables that are parameterizing the target language structure, the variables are substituted in to the structure. In the label keyword the expression '`%=A%`' means that the content of the variable A will be substituted by the result of the generation of the keyword inside the `parameter c`. You may wonder why the subroutine is nested within the Label. The generator returns a user interface control. In Visual Basic definition of the user interface and the accompanying code is seperated, just like in Java the definition of datamembers and methods is seperated in a class definition. The two AST nodes label and sub may thus not be returned together for example in a single set. To solve this problem, other target structures (AST nodes of the target language) can be hooked to the actual target structure that is returned. The former is called a non-local target structure. The hooking is achieved by nesting the two structures.

```
keyword label {
     ast { public class AST { keyword c; } }
     recognition pattern { ("label", keyword c) }
     generator {
          public VB.Label generate() {
                VB.Expression A = generate c;
                return #VB.Label{
                   Begin VB.Label
                         name = "mylabel"
                   End
                   #VB.Sub{
                       Public Sub Form_Load()
                             mylabel.caption = %=A%
                       End Sub
                    }
                };
          }
     }
}
```

3.3 Language Specification

The language specification *glues* the keywords together and *provides information about the overall structure of the language* to the keywords. Below the language specification (LS) for our initial toy language. LS are written in XML syntax. It composes two keywords: a string and a label so that the caption of the label is a string. The `keyword` tag introduces a keyword into the language. Inside this tag other tags can be written to configure the keyword, e.g. the `param` tag. The first line introduces the label keyword (defined above). It has one parameter `c` in which the `string` keyword is put. This is how the formal parameter `c` defined in the definition of the `label` keyword gets bound to the `string` keyword.

```
<keyword type="label">
<param name="c">
       <keyword type="string"/>
</param>
</keyword>
```

In KBP (like most other environments) a language implementation consists of two parts: a parser that constructs the AST tree from the DSL program and a transformer to translate this tree to the AST tree of the desired program written in the target language. The KBP development environment generates a parser and a transformer out of the language specification. The parser is built by composing the recognition patterns of the keywords according to the language specification. The transformer is built likewise using the generators of the keywords. After parsing, the transformation is initiated by triggering the generator belonging to the keyword of the toplevel AST node of the AST tree.

In the next two subsections we will discuss the parameterization mechanism of keywords, how these parameters can get values in the language specification and how keywords are scoped through the language specification.

3.4 Information Exchange

The information non-local to a transformation (e.g. the UI language of the translate-transformation) is supplied via the language specification. This information is thus parameterizable, hereby reducing coupling of the keyword with the overall language structure.

To illustrate how in KBP non-local information is obtained we will likewise extend the current language specification with a new `translate` keyword just like we did in the problem statement. The `translate` keyword is parameterized with a keyword-parameter that contains the `string` keyword, and a value to provide the UI language. The latter can be provided to the keywords by means of the value-tag (see the example below). Using this mechanism we are free to determine how to provide the non-local information, the form and the location of the non-local information. One is free to chose how the information can be provided, e.g. a location in the AST tree or a fixed value `<value ...> "english" </value>` etc. The form denotes which keywords hold the information needed. The location is the location that the information containing keyword(s) have in the grammar.

To locate and retrieve information from other parts of the AST tree a path can be defined between two nodes. In KBP every keyword is a Document Object Model [Whi02] (DOM) element and the AST is thus a DOM tree. This gives us the advantage of being able to reuse many algorithms already available in the java programming language api. One of them was *xpath* [AB02]. With xpath expressions, connections between two or more nodes can be easily established (symbolised by the arrows in figure 1).

To avoid the problems caused by coupling of DSL components to a specific DSL language in which they are used, we specify where the context-information of the component is located with *structure shy paths* [Lie96], [LPS97]. These paths are more robust to changes in the DSL, because they do not contain detailed information about the actual path that needs to be followed to reach the desired AST node. It turns out that xpaths can be used to specify structure shy paths. For example the path //XXX denotes a grand child XXX regardless of the position of the child in the current subtree. These paths are more robust to changes in the specification language than fully specified paths; and thus increase the evolvability of the domain specific language.

```
<keyword type="label">
<param name="c">
        <keyword type="translate">
        <param name="value">
                <keyword type="string"/>
        </param>
        <value name="language" type="String">
                execute("/ancestor::XXX//Language/Property")
        </value>
        <value name="translator" type="String">
                "translator"
        </value>
        </keyword>
</param>
</keyword>
```

The above code shows the language specification for the DSL of the component library that supports multilinguality. The `translate` keyword is put inside the `c` parameter of the label and encapsulates the `string` keyword. The `translate` keyword is defined below. The information which is provided through the value-tags in the language specification are accessible within the transformation with getter-methods. In the example, the values in the value-tags with the name `language` and `translator` accessible via respectively the `getTranslator()`

method and the getLanguage() method. Inside the value tag a path /ancestor ::XXX//Language/Property to another AST node is executed via the execute method. The first part of the path /ancestor::XXX locates an ancestor AST node with the name XXX. The second part of the path //Language searches for a Language AST child node somewhere located in the XXX AST node. And finally the third part of the path /Property retrieves the direct Property AST child node of the Language AST node.

In this example no information needs to be passed from the keyword to the language specification, but there is support for it. Making two-way communication possible.

```
keyword translate {
     ast { public class AST { keyword value; } }
     recognition pattern { keyword value }
     generator {
          public VB.Label generate() {
                Object A = generate value;
                return #VB.Expression{
                   %=getTranslator()%.translate(
                          %=A%,
                          configfile.getProperty(%=getLanguage()%))
          }
     }
}
```

With this approach the non-local information for a transformation is no longer hard-coded in the transformation itself but is supplied via the language specification. This information is thus parameterizable, and reducing coupling of the keyword with the overall language structure. Furthermore, the code to retrieve and locate information out of the AST defined in the language specification is structure shy, making this code more robust to changes in the language.

3.5 Composition of Transformations

In the previous section we've shown that keywords can be parameterized with other keywords and with configuration information. Although the composability is already greatly improved we identified another important issue concerning the composability of the transformation being the scope of a transformation.

The scope of a transformation is the definition of the regions in the AST tree which will be transformed by the transformation. In KBP the scope of a keyword is defined by its position in the language specification. This is illustrated with the translate keyword whose scope is defined through the wrapping around the string keyword and the placement inside the label keyword (cfr. the language specification above). Therefore the transformation in translate keyword doesn't need to specify where in the source AST tree it must be applied, rendering the keyword more independent and the domain language more evolvable.

Lets revise the extension of the GUI library and DSL from the problem statement to illustrate how the definition of the scope of transformations in KBP increases the evolvability of the DSL. In the problem statement the language has been further extended to support forms i.e. containers for labels. The titles of the forms must also be translated. The extension in KBP is pretty straightforward. The form can be easily added to the language and thereby reusing the both parts of the translate keyword, e.g. the bnf rule and the transformation respectively the recognition pattern and the generator. Because the scope of a keyword is defined in the language specification and there is no further coupling between the translate keyword and overall language structure, the translate keyword could be easily reused. Therefore, in contrast with the situation in the problem section, only the language specification must be changed.

Below is the part of the language specification to support multilingual forms. Recall that a form has a name and a title. In KBP the form will thus have three parameters `name`, `title` and `body`. The `title` parameter has been bound to the `translate` keyword which encapsulates the `string` keyword. The `name` parameter has been bound to an identifier. The `body` parameter is bound to the label keyword which has been defined earlier.

```
<keyword type="form">
<param name="title">
        <keyword type="translate">
        <param name="value">
                <keyword type="string">
        </param>
        <value name="language" type="String">
                execute("/ancestor::XXX//Language/Property")
        </value>
        <value name="translator" type="String"> "translator" </value>
        </keyword>
</param>
<param name="name"> <keyword type="ID"/> </param>
<param name="body">
        <keyword type="Label">
        ...
        </keyword>
</param>
</keyword>
```

Recall that the `translate` keyword must be parameterized with the UI language . The code to retrieve the UI language from the AST was defined using structure shy paths. When the path should become invalid in the context of a form the translate keyword can be easily reused and be redefined, as is presented in the language specification above. When the path remains valid in the context of a form, then the `translate` keyword must not be redefined but can be referenced a such. This is shown in the code below where the keyword in the parameter title points to the keyword with id `translatedstring`. In either case the DSL is easily evolvable and the `translate` keyword is reusable.

```
<keyword type="form">
<param name="title">
       <keyword refid="translatedstring"> </keyword>
</param>
...
</keyword>
```

The following illustrates the usages of the form extension in a DSL program.

```
form form1 title "sample form" {
        label "title"
}
```

The definition of the `form` keyword is given below. The recognition pattern and the AST node definition are obvious. The generator creates a form AST node `f` and return this as its result. First the keywords in the parameters `name`, `title` and `body` are generated. A Form definition contains two parts: a UI part and some code. The name, title and the body are parameterizing the forms definition consisting of a form UI definition and a subroutine `Form_Load`. The construction and parsing of the AST form node is done by the expression `#VB.Form{ ... }`). The `label` keyword (when bound to the body parameter through the language specification) returns a label AST node which contains a subroutine AST node. The result is stored in the variable `C`. The content of the variable is inserted in the UI part of the form, a part where only user interface controls may be inserted. Therefore the subroutine AST node is automatically extracted form the label AST node and placed in the appropiate syntactical area.

```
keyword form {
     ast { public AST {
              keyword name;
              keyword title;
              keyword body;     }
     }
     recognition pattern {
         ("form", keyword name, "title",
          keyword title, "{", keyword body, "}" )
     }
     generator {
          public VB.Form generate() {
                String A = generate name;
                String B = generate title;
                Object C = generate body;
                return #VB.Form{
                   Begin Form
```

```
                    name = %=A%
                    %=C%
                End Form
                Public Sub Form_Load()
                    %=A%.title = %=B%
                End Sub
            };
        }
    }
}
```

4 Discussion

The keywords in KBP are stand-alone, more reusable and composable language components. Stand-alone because in their definition they contain their syntax representation, AST node to hold information and a generator to implement its effect. Due to the parameterization of keywords with information that is non-local to them keywords are reusable in the sense that when the language in which they are used evolves they remain usable as such. Keywords are easily composable because they are parameterizable with other keywords and their scope is defined through the language specification. Changing the language involves only changing the composition and parameterization of the keywords in the langauge specification.

The DSL is implemented via a language specification that configures, composes and glues the form, label and translate keywords. The following code is an example program written in the DSL:

```
language "config.ini" UILanguage
form form1 title "sample form" {
        label "title"
}
```

which gets translated to

```
Begin Form
     name = "form1"
     Begin VB.Label
           name = "mylabel"
     End
End Form
Public Sub Form_Load()
     form1.title = translator.translate("sample form",
                              configfile.getProperty("UILanguage"))
     mylabel.caption =  translator.translate("title",
                              configfile.getProperty("UILanguage"))
End Sub
```

A first prototype of KBP has been implemented in Java supporting all the features and properties introduced in this paper. In addition to those, more experimental features concerning the integration of the results produced by each keyword into a AST, providing defaults values to keywords by reusing other keywords etc. are included. Various DSLs have been implemented, e.g. a metacircular implementation of the system, tuple calculus, small business administration programs, etc. ranging from 10 to 45 keywords.

5 Related work

The architecture of KBP is quite similar to the one of intentional programming [Sim96], [Sim95] and delegating compiler objects (DCO's) [Bos97]. In both environments DSLs are built out of the composition of language components. However, the granularity of DCO's is more corse grained then keywords. DCO's are actually small conventional compilers which contain lexers, parsers, transformers etc.

The language specification mechanism in KBP is unique. Only the Jakarta Tool Suite (JTS) [BLS98] and DCO's use BNF as a, somewhat limited composition mechanism since it does not allow to configure the parameters of the transformations. This language specification is used to compose, trigger and scope the transformations. Environments without this use some sort of other scheduling and scoping mechanism. The transformations in ASF+SDF Meta-Environment [vdBK02] and XSLT [Cla99] define within themselves which elements of the AST they can transform, rendering them tightly coupled, highly dependent and not reusable in other parts of the language. In intentional programming dependencies can be defined to further aid the scheduling mechanism. Jargons [NJ97], [NAOP99] do not have a language specification, nor sheduling, nor scoping mechanism. A transformation is linked to a syntax description and is triggered where the user has used this syntax.

In each development environment mentioned above there is no explicit support for parameterizing the transformations. In most cases some work-around is possible because either the transformations are written in a GPL allowing proprietary solutions, or either some other feature can be bent to serve for this purpose.

Only in XSLT non-local information between the transformations or language components can be easily exchanged via structure shy navigation expressions. Both KBP and XSLT use XPaths to accomplish this. But due to lack of parameterization in XSLT these expressions are hard coded in the transformations themselves. Intentional programming and JTS does offer some navigation primitives but the resulting code contains detailed information about the structure of the AST limiting severely the evolvability of the DSL language. Retrieving non-local information is not supported in Jargons and in the ASF+SDF environment.

6 Conclusion

Domain specific languages are an excellent mechanism to encapsulate a library of components and hide the complexity of component composition from the library user. However as component libraries evolve, so must their DSLs, which represents an unacceptable cost. In this paper we introduced keyword-based programming as a technology to develop more easily evolvable domain specific languages. Using this technology, developing DSLs for immature libraries has become more feasible.

Keyword-based programming extracts these parts that cause the entanglement from the transformations with the overall language structure and other transformations and introduces a separate language specification which glues the language implementation 'keywords' together. This separation allows to write language features (keywords) that are not tangled with context, non-local and scope information by allowing them to parameterized with configuration information and other keywords. Instead, these are provided by the language specification. The capabilities of KBP allow a developer to implement a DSL which is more easy to evolve and, as such, is more suitable to write a DSL for continously evolving component libraries.

Structure shy paths are used to exchange information between the various parts of the language (the keywords) allowing the language specification to evolve without invalidating those paths.

7 Acknowledgements

I would like to thank Johan Brichau and Tom Tourwe for comments on this paper.

References

[AB02] Don Chamberlin Mary F. Fernandez Michael Kay Jonathan Robie Jrme Simon Anders Berglund, Scott Boag. Xml path language (xpath) 2.0 w3c working draft 15 november 2002, 2002.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.

[Bos97] Jan Bosch. Delegating compiler objects: Modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1):66–92, Spring 1997.

[BST$^+$94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software system generators. In *IEEE Software*, pages 89–94, September 1994.

[Cla99] James Clark. Xsl transformations (xslt) version 1.0 w3c recommendation 16 november 1999, 1999.

[Fal01] David C. Fallside. Xml schema part 0: Primer w3c recommendation, 2 may 2001, 2001.

[FNP] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. pages 243–256.

[GS94] S. Wingo G. Shepherd. *MFC Internals: Inside the MFC Architecture*. Addison-Wesley, 1994.

[JBP89] J.Heering J.A. Bergstra and P.Klint. Algebraic specification. *ACM Press/Addison-Wesley*, 1989.

[LB00] M. Glandrup M. Aksit L. Bergmans, B. Tekinerdogan. On composing separated concerns, composability and composition anomalies. In *ACM OOPSLA'2000 workshop on Advanced Separation of Concerns, Minneapolis*, October 2000.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[LPS97] K. Lieberherr and B. Patt-Shamir. Traversals of object structures: Specification and efficient implementation, 1997.

[MM99] Jan Bosch Michael Mattsson. Experience paper: Observations on the evolution of an industrial oo framework. *ICSM*, pages 139–145, 1999.

[NAOP99] Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olsen, and Paul M. Pontrelli. Jargons for domain engineering. In *Domain-Specific Languages*, pages 15–24, 1999.

[NAU60] Peter NAUR. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[NJ97] L. Nakatani and M. Jones. Jargons and infocentrism. In *First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 59–74, 1997.

[SB97] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[Sim96] C. Simonyi. Intentional programming - innovation in the legacy age, 1996.

[Sin96] Vivek P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators*. PhD thesis, 1996.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[TB00] C. M. Sperberg-McQueen Eve Maler Tim Bray, Jean Paoli. Extensible markup language (xml) 1.0 (second edition) w3c recommendation 6 october 2000, 2000.

[vdBK02] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*. Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, July 2002.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357–??, 2001.

[Whi02] Ray Whitmer. Document object model (dom) level 3 xpath specification w3c working draft 28 march 2002, 2002.