GU-DSL – A Generic Domain-Specific Language for Data- and Image Processing

Thomas Högg[1], Günther Fiedler[1], Christian Köhler[2], and Andreas Kolb[2]

[1] Christ-Elektronik GmbH, Alpenstr. 34, 87700 Memmingen, Germany
thoegg@christ-elektronik.de, gfiedler@christ-elektronik.de
[2] University of Siegen, Hölderlinstr. 3, 57076 Siegen, Germany
christian.koehler@uni-siegen.de, andreas.kolb@uni-siegen.de

Abstract. The complexity of image processing tasks has risen during the last years. To counteract this trend, we introduce the novel Domain Specific Language GU-DSL allowing to shorten the time-to-market and improving the development process. Therefor we use some base concepts of Java and C#, adopting the idea of encapsulating classes and flow-models using diagrams as e.g. done by UML, but in a textual form. Giving developers the freedom of individual modeling, the DSL forces the developer, to adhere to special structures and requirements using flow modeling which helps reducing recurring mistakes.

Keywords: Domain-specific languages, Modeling, Computer Graphics and Vision

1 Introduction

The complexity of image processing tasks has risen during the last years for several reasons. On one side, more and more industrial machines profit from image processing support, e.g. by using a camera system for visual inspection. On the other side, camera systems and the processing components (as e.g. desktop PCs) became better and cheaper. The complexity and the demand for a short time-to-market requires improving development processes. This can be achieved using different methods. One method can be the usage of graphical modeling tools as e.g. UML [5] or LabVIEW [5]. An other interesting method is the usage of Domain Specific Languages (DSL) allowing the design of novel problem related programing and description languages [2]. In this technical report, we will introduce a novel DSL based on Eclipse Xtext. It is developed from scratch, but using some concepts from Java and C#. Furthermore, it is adopting the idea of encapsulating classes and flow-models using diagrams as done by UML, but in a textual form. This has the advantage to give developers the freedom of modeling as he sees fit. But at the same time, the DSL forces the developer to keep special structures and requirements using flow modeling. This helps reducing recurring mistakes. The DSL is specially designed to make the model-to-text transformation as easy as possible and to support C++ as intermediate textual representation.

We provide the following novel DSL features in this technical report:

– A diagram based, object oriented, textual modeling language
– Class-Diagram support
– Activity-Diagram support
– An expression language allowing to implement sequential code sections within classes, class-methods and activity-diagram nodes

The remainder of this paper is organized as follows. Sec. 2 discusses the related work while Sec. 3 introduces our new DSL. Sec. 4 concludes this paper.

2 Related Work

Especially related to our presented approach is the work of Efftinge et. al [2]. They present Xbase as a reusable part of Xtext, allowing the usage of expressions in a control flow. Furthermore, they provide full interoperability between Java and Xbase based DSLs. This gives engineers the possibility to start the development of new DSLs from a generic base. Our approach uses parts of the Xbase Extended Backus Naur Form (ENBF) notation.

In 2010, Axelsson et al. [1] presented the DSL Feldspar, allowing the high-level modeling of digital signal processing (DSP) algorithms. While using data-flows for higher level programming, it is embedded in Haskell and uses some optimization techniques for code generation.

Another DSL for big data analysis for telecom industries was introduced by Senbalci et al. in 2013 [6]. It abstracts and simplifies processing to a higher level big data solution system called Petaminer.

Fisher et al. introduced PADS in 2005 [3]. It is a DSL for processing of ad hoc data allowing to directly describe the physical layout and semantic properties of data. A compiler generates all the required tools and libraries for data manipulation.

Besides the previously shown related DSLs, many other DSLs exist in the domain of data description, processing and analysis. But there is no complete image processing DSL available at this time.

3 GU-DSL – A Generic Domain-Specific Language

The following section will introduce the main important features of GU-DSL, allowing the implementation of image- and data-processing algorithms.

3.1 Structural Modeling using Class-Diagrams

Modern object oriented programming languages in general use classes and namespaces to form an abstraction of real-world objects. This allows structuring software by encapsulating functionality belonging together. As long as there are only a few number of classes or objects, it is difficult to keep the overview. Class-diagrams are a good tool to model complex structures. While allowing the design of individual class structures (variables and methods), they also provide the possibility to visually show object interconnections as associations or inheritance between related classes.

We adopt this concept and allow the definition of classes, interfaces and attributes (used to visualize meta-information). Listing 1 shows the definition of a simple diagram with an abstract image class, an example for class inheritance and interface implementations. This is in general well-known from languages as C# and Java and forms the base of our system.

```
ClassDiagram ExtendingTypes
{
  // Interface definition
  public interface IImage
  {
    public bool load(string filename);
    public bool save(string filename);
  }

  // Interface implementation
  public abstract class Image implements IImage
  {
    public int width;
    public int height;

    public bool load(string filename);
    public bool save(string filename);
  }

  // Class Extension
  public class ExtImage extends Image
  {
    public void filterNoise();
    public void smooth();
  }
}
```

Listing 1 Extended class diagram showing interfaces and class inheritance.

Besides classes, enumerations are another important kind of structure allowing value grouping (Listing 2).

```
enum ImageFormat
{
  Format_RGB32 = 2,
  Format_ARGB32 = 3,
}
```

Listing 2 Definition of an enumeration.

Enumerations can be used as independent types as the built-in types **byte**, **int**, **float**, **real**, **string**, **bool** and **void**.

Method parameters and fields can have qualifiers applied. This gives the possibility to restrict or to grant access rights and define the visibility (Listing 3).

```
// A constant, public field
public const int i = 0;
// A protected reference field
protected ref real j;
// A private static field
private global int g = 100;
// A public constant reference to a static field
public global const ref int k = ref g;
// A public static method
public global void memcpy(ref void dst, ref void src,
int numBytes);
```

Listing 3 Field, method and parameter access qualifiers and visibility.

To allow modeling of meta-information, e.g. to support a more flexible code generation, we introduce attributes (comparable to C# attributes and Java annotations, see Listing 4).

```
ClassDiagram Attributes
{
  public attribute CppType
  {
    public string name;
  }

  public attribute GeneratorVisibility
  {
    public bool visible = true;
  }
}
```

Listing 4 Definition of generator attributes.

Attributes can be used on classes, methods, fields and enumerations (see Listing 5).

```
[Attributes.GeneratorVisibility(visible = false)]
public abstract class Image{}
```

Listing 5 Definition of generator attributes.

Until now, we can only define the general structure of all the necessary objects. Using associations well-known from the graphical UML modeling, we allow the developer to define relations between objects. This is simply done by defining fields in classes, as can be seen in Listing 6.

```
// Image contains 0 .. to n sub images
self aggregation [0 .. *] subImages : Image;

// At least one or more pixel belong to an image
self composition [1 .. *] pixel : Pixel;

// Bidirectional association: Image contains
// 0 to n pixel. One pixel belongs to one image
self [1] contains : Pixel [0 .. *];

// A uni-directional association: A pixel
// has a parent, but the parent does not know it
self parentImage : Image;
```

Listing 6 Association types.

Besides simple classes, interfaces and enumerations, we support class- and interface-templates as well as reference types. Using these definitions, we are able to exemplary model a complete structural representation of image- and data-processing algorithms.

3.2 Definition of Behavior Modeling

The second important part is the definition of object and system behavior. UML provides several kinds of diagrams (activity-, state-machine or sequence-diagrams). Our approach uses two incorporating methods. The first one uses an expression language (similar to XBase [2]), allowing sequential coding and lowering the barrier to using our presented approach. A second method uses an extended activity-diagram allowing flow-modeling. The diagram fully supports and incorporates the expression language to solve the main drawback of rapidly rising visual complexity of pure activity-diagram modeling, as stated in the introduction.

3.3 Behavior Modeling using Expressions

As mentioned previously, our expression language uses the base concepts of XBase and other well-known programming languages. It is based on the types introduced in Sec. 3.1 and is specially designed in a way to make the text-to-text (T2T) transformation (code generation) as easy as possible. It has special domain-specific extensions, e.g. references to improve algorithm performance. The next sections will introduce the most important expression features.

Variable Expressions The expression grammar allows three types of variables declarations:

1. Mutable variables, defining variables that can be changed:

```
var int i = 0;
var Image image;

```

2. Constant variables, defining variables that cannot be changed:

```
const int j = 0;

```

3. Reference variables, as known from C/C++ as pointers, allowing direct access to memory addresses:

```
var ref int k = 0;
var ref Image image;

```

Reference variables are treated in a special way, different from the C/C++ dereferencing mechanism. Depending on the assigned value type, the kind of assignment is automatically chosen (see Listing 7).

```
var int i = 0;              // Declare variable i
const int j = 0;            // Declare constant j
var ref int k = ref i;      // Assigning i as
                            // reference to k
k = 100;                    // Assigns 100 to k,
                            // and respectively to i
k = j;                      // Assigning the content
                            // of j == 0 to k
k = ref j;                  // Assigning the j as
                            // new reference to j
```

Listing 7 Reference assignment and automatic dereferencing.

Using this mechanism simplifies the language usage and improves maintainability. Additionally, through the support of references, it is possible to develop fast code, which is the most important requirement in image processing algorithm development.

Loop Expressions Loops are a central part of programming languages, allowing simple repetition of statements. We support three kinds of control structures.

1. Head-controlled loops:

```
loop(i < 100) { i = i +1; };

```

2. Tail-controlled loops:

```
do { i = i + 1; } loop(i <= 100);

```

3. Conditionally head-controlled loops:

```
for(i = 0 : 1 : i < 100) { };

```

Conditional Expressions Besides loops, we also support two kinds of conditional expressions:

1. If-Else-expressions:

```
if(i < 100) { k = 0; }
else{ k = 1; };

```

2. Switch-Case-expressions:

```
switch i:
{
case 0: {}
default: {}
};

```

Other Expressions The previous sections introduced the most important expressions of our language. But the language also supports basic expressions as assignments ($=$), equality checks ($==, !=$), logical operations ($\&\&, \&, ||, |$), comparisons ($>=, <=, >, <$) and expression grouping in blocks ($\{\}$) that are used in the same way as in C/C++.

All the expressions previously described can be attached to class methods to allow behavior modeling using expressions. Furthermore, we allow, using the grammar in activity-diagram nodes introduced in Sec. 3.4.

3.4 Behavior Modeling using Activity-Diagrams

The previous section introduced our expression language allowing simple textual behavior modeling. This section shows how to use textual activity-diagrams to give the developer the possibility to abstract algorithms or behavior in a more object oriented manner. Activity-diagrams provide an easy way to model data flow or single system parts. They support concurrency, conditional decisions and also loops. We extend the UML activity-diagram by the following features to make them more reusable:

- Class-assignments
- Diagram input parameter (as in UML 2.x)
- Activity-diagram variable definitions
- Action-node definition
- Expression support within nodes

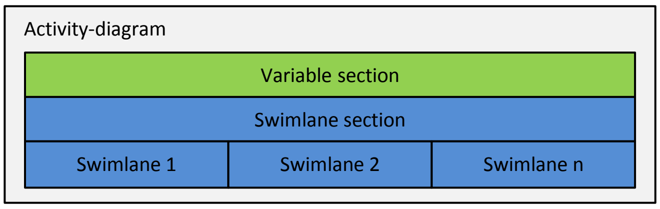

Fig. 1: The two sections of an activity-diagram.

As can be seen in Fig. 1, an activity-diagram consists of a variable section defining local variables and an arbitrary number of concurrent swimlanes. Our approach assumes that an activity diagram describes the flow within a class method and is called as shown in Listing 8.

```
call behavior AdMemberAccess("C:\\example.png",
                             "C:\\example_filtered.png");
```

Listing 8 Call of an activity-diagram.

Listing 9 shows an example diagram containing important diagram features, which are described in the next sections.

```
ActivityDiagram AdMemberAccess(string filenameLoad,
                               string filenameSave)
{
  // Diagram specific variable declarations
  public int i = 0;

  swimlane Swimlane1, owner ExtendedBaseTypes.Image
  {
    start S1 { => load(filenameLoad); }

    // The action represent the load method
    // of the Image class
    action load(string name)
    {
      i = 0;
      => filterImage;
    }

    // A simple action-node doing some filtering
    action filterImage
    {
      i = i + 1;
      // Do some filtering

      // Recursive node call ==> lowwop
      [i < 10]  => filterImage;
      // Call the save method
      [i >= 10] => save(filenameSave);
    }

    // The action represent the save method
    // of the Image class
    action save(string name)
    {
      => Finish;
    }

    final Finish

  }
}
```

Listing 9 An example activity-diagram.

Class-assignments Our implementation allows assigning owner classes to either a diagram or to the individual swimlanes. This offers the possibility to directly interact with classes. It provides access to all class member variables and methods as can be seen in Listing 9 (load- and save-node).

Diagram input parameter The UML defines activity diagram input and output parameters as well as objects, which is an essential extension to allow the re-usage of diagrams. Listing 9 shows the re-usage of these parameters as object flow between nodes.

Activity-diagram variable definitions Another newly introduced feature is the usage of local variable definitions. Class-assignments allow access to class member-variables. But in most cases, this isn't enough. We allow the definition of local variables that can be used in the same way as member variables for transition guards and also within expression statements.

Action-node definition Action-nodes are the main modeling nodes. They provide the basic functionality and allow visualizing all kind of problems. The connection between nodes is realized as either guarded or non-guarded transitions. This provides the possibility to model conditional and loop flows, but most of the time it is complicated and confusing. UML has already introduced special kinds of nodes such as forks, joins and decisions. But from our point of view, it is not enough to provide a simple programming interface. Our additional and extended node types are introduced in the following sections. Furthermore, we allow three different kinds of action-node declarations. The first one represents a method call of an assigned class (e.g. load or save in Listing 9), while the second method (filterImage) shows the possibility of defining collections of expression statements. The third possibility is defining action-nodes that are assumed to be a method call, but they don't have to be members of a class. Having the same signature as method-call nodes, they are treated in a special way by the code generator. This allows the reduction of generated code by reusing them in a second occurrence as a simple method call.

Other node-types Besides the introduced new nodes, we also support **start-**, **final-**, **decision-**, **fork-** and **join**-nodes.

3.5 Summary

This section has shown the most important specifications of GU-DSL, which allow for modeling of computer-vision algorithms and are also the basis of our planned graphical modeling framework. Required class- and activity-diagram-declarations and also expressions have been introduced and their basic usage was demonstrated.

4 Conclusion And Future Work

In the previous sections we introduced a new domain-specific language, allowing developers and modelers to design data and image processing applications. The modeling toolchain to design the DSL is based on Eclipse-Xtext to design the DSL. In combination with our code generator and the high level abstraction C++ framework, it is possible to easily develop image processing filters, e.g. a bilateral filter or even more complex processing methods.

In the future, we plan to extend the language by graphical editors and other important diagram types such as object-, state-machine- and component-diagrams. Especially component diagrams allow the re-usage and encapsulation of recurring image processing problems while reducing implementation time. In combination with the work proposed in this paper, the quality of such software systems and their time-to-market can be improved significantly. Furthermore, we plan to add a full validation system using OCL (Object Constraint Language) in combination with EVL (Epsilon Validation Language) [4]. This can additionally improve the quality by the interactive help for the developer showing problems directly during design. Besides development support, we also want to extend the system by an interpreter and the possibility to use the Eclipse Debugger Support via GNU Debugger (gdb).

5 Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the research training group GRK 1564 'Imaging New Modalities'.

References

1. Axelsson, E., Claessen, K., Devai, G., Horvath, Z., Keijzer, K., Lyckegard, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on. pp. 169–178 (July 2010)
2. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: Implementing domain-specific languages for java. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. pp. 112–121. GPCE '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2371401.2371419
3. Fisher, K., Gruber, R.: Pads: A domain-specific language for processing ad hoc data. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–304. PLDI '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1065010.1065046
4. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Rigorous methods for software construction and analysis. In: Abrial, J.R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis, chap. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pp. 204–218. Springer-Verlag, Berlin, Heidelberg (2009)
5. OMG: Uml (November 2014), http://www.uml.org/
6. Senbalci, C., Altuntas, S., Bozkus, Z., Arsan, T.: Big data platform development with a domain specific language for telecom industries. In: High Capacity Optical Networks and Enabling Technologies (HONET-CNS), 2013 10th International Conference on. pp. 116–120 (Dec 2013)