Abstracting Data and Image Processing Systems using a Component-based Domain Specific Language

Thomas Hoegg, Christian Koehler and Andreas Kolb
Computer Graphics and Multimedia Systems Group, University of Siegen, Hoelderlinstr. 3, 57076 Siegen, Germany

Keywords: Modeling, Components, Domain-specific Languages, Image Processing.

Abstract: This work proposes a textual and graphical domain-specific language (DSL) designed especially for modeling and writing data and image processing algorithms. Since reusing algorithms and other functionality leads to higher program quality and mostly shorter development time, this approach introduces a novel component-based language design. Special diagrams and structures, such as components, component-diagrams and component-instance-diagrams are introduced. The new language constructs allow an abstract and object-oriented description of data and image processing tasks. Additionally, a compatible graphical design interface is proposed, giving modelers and architects the opportunity to decide which kind of modeling they prefer (graphical or textual, including round-trip engineering).

1 INTRODUCTION

The requirements on software systems were rising dramatically during the last years. Especially the usage of complex software architectures for technical applications, such as image processing or similar (sensor-) data processing tasks, has heavily increased due to cheaper and more powerful hardware. While system requirements become more and more complex, the time to market should be shortened and the software quality should be improved at the same time. Setting up new data and image processing systems is an always recurring task. Having predefined domain-specific languages (DSLs) and tested generic default runtime architectures, supporting functionality as Graphical-User-Interface (GUI) and algorithm interaction can catalyze development. As an add-on on top of a data processing runtime, component-based software engineering (CBSE) can help separating data processing problems and algorithms into packed, reusable software components. After designing such a component, engineers and developers are able to instantiate it and connect it to other components using predefined interfaces. However, one of the biggest drawbacks in the domain of data and image processing is that no real standard architecture has been established yet.

During the past years, several techniques have been established in software engineering but are rarely used for image processing. The two most important approaches are graphical modeling and DSLs. Graphical modeling gives modelers the possibility to create a relatively simple, high-level, iterative graphical architecture design, where all relevant parties (programmers and non-programmers as e.g. project leaders) can have an abstract but graphical view on the system in development. This can help with identifying and solving problems at an early stage. DSLs are the second important approach. While the concept of DSLs has been known for a long time, they have gained much importance in the last years due to tools simplifying the language development, e.g. Xtext (Efftinge et al., 2015). DSLs are divided into two types: internal and external DSLs (Fowler, 2010). Internal DSLs use the existing infrastructure of host programming languages. Examples for such DSLs are OpenCL or OpenGL. Both languages are a C dialect, extending C to their requirements. External DSLs however are designed from scratch after a full analysis of the problem description. Using special keywords, abstractions and control structures, they are able to illustrate complex problems mostly in simpler and more compressed forms. A big drawback is that the whole infrastructure such as parsers, lexers and compilers has to be built. Well-known examples are the Unix shell scripts or SQL.

Both kinds of approaches, graphical modeling and also external DSLs, are unfortunately hardly used in the domain of image processing and computer vision. Developers mainly want to concentrate on algorithm

Hoegg, T., Koehler, C. and Kolb, A.
Abstracting Data and Image Processing Systems using a Component-based Domain Specific Language.
DOI: 10.5220/0005743502920300
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 292-300
ISBN: 978-989-758-168-7

development and not on software design issues. This is why we propose GU-DSL, an external data and image processing related language, combining graphical and textual modeling to reduce architecture and system design times. For this purpose we contribute the following novel main features:

- A language concept and implementation of components using ports and interfaces based on Xtext
- Class-, component- and novel component-instance-diagrams for component and system design
- A textual and graphical tool infrastructure using Eclipse Xtext and GMF

The remainder of this paper is organized as follows: Sec. 2 explains the newly proposed language and software architecture concept. In Sec. 3, we show our novel DSL features. Sec. 4 exhibits an image processing modeling example. In Sec. 5, the related work to this paper is discussed. Sec. 6 concludes this paper.

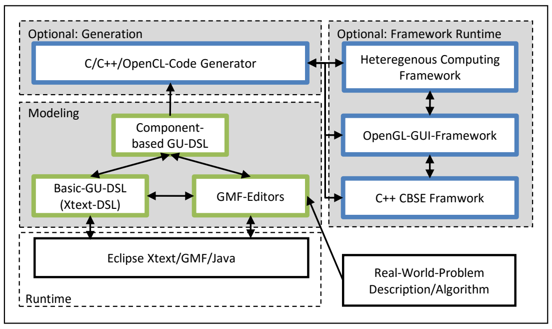

Figure 1: The principle schema of the modeling framework.

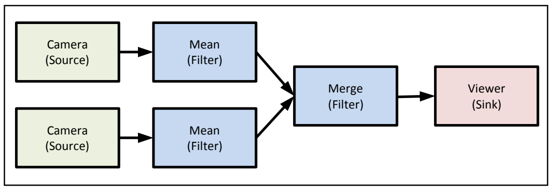

Figure 2: An image processing example using several components.

2 SYSTEM CONCEPT AND OVERVIEW

GU-DSL is a diagram based, object oriented modeling language supporting structural- and behavior-modeling in a textual and graphical, model driven way. Using a Java and C# like syntax, it is easy to learn and offers an abstract way for textual modeling of especially data and image processing tasks. Apart from using class- and activity-diagrams to model objects and system behaviors, it gives architects the possibility for sequential behavior coding using an embedded expression language. The DSL's activity-diagrams and the expression language are just mentioned for completeness and are not part of this paper.

The paper proposes a novel component-based language in a textual and graphical form. For component-oriented modeling, we pick up the basic concepts specified in literature as interfaces, classes, components, ports and interface connections and integrate them in a novel way into our DSL. Hence, we add special novel keywords, structures, component implementations and communication concepts that we will show in the following sections.

Fig. 1 shows the principle setup of the framework. The contributed parts with this paper are highlighted in green. Three collaborating system parts are comprised: modeling, code-generation and the framework runtime. The basis of our system is built in Eclipse in combination with some modeling plugins such as Xtext (a DSL generation language) and GMF (Graphical Modeling Framework). On top of Eclipse, there is our DSL-modeling environment. It contains three interacting parts: the basic DSL (class- and activity-diagrams, expression language), the component-based language parts and the GMF editors allowing for graphical modeling. By using the modeling part, the system architect can easily abstract real-world problems and algorithm descriptions and use the model as input for the future code generator.

Data and image processing tasks can rapidly grow in complexity. The abstraction towards reusable components helps handling this. Having a look on such a processing system (see our example in Fig. 2) shows, that we generally have a pipeline system starting with a source (often a camera, sensor or a database), continuing with filters (e.g. noise reduction or data evaluation) and finishing with sinks (e.g. showing or storing data). In software-engineering, this problem description can be mapped by the *pipe and filter architectural pattern*, allowing pipeline based data processing. It serves as a basis for our novel language concept as we will show in the next sections. A detailed description of the particular DSL parts can be found in Sec. 3.

Components form the basis of our system. Interfaces, known e.g. from UML and SysML, allow communication between components. The direction depends on the interface definition and can be either uni- or bi-directional. The DSL is defined with the following restrictions:

- **Sources:** Outgoing communications (uni-directional)

- **Filters:** Incoming and outgoing communications (bi-directional)
- **Sinks:** Incoming communications (uni-directional)

Ports are used as flexible input and output points of components by providing required interfaces. Every port is uniquely assigned to a component and can be connected to other valid ports, allowing a predefined, direct communication as we will see in the later sections.

3 THE COMPONENT BASED ENGINEERING LANGUAGE

GU-DSL is a language especially designed for object-oriented modeling of data and image processing problems. Different to other languages in this domain, it is designed in a way that model driven problem solutions, both textual and graphical, are the main focus. Other languages often concentrate on the compression and simplification of code only.

This section shows all the features required for understanding our concept and novel ideas. Components are a well established concept in software engineering, however, the novel contribution of this approach is the way how all the different diagrams (class-, activity- and component-diagrams) and language features interact to reduce future errors and to speed up development. Especially the way how components interact with classes, types and methods using behavior-diagrams or sequential coding are unique in this domain and will be shown in the following section. Additionally, the proposed component approach can be used standalone for architectural design and modeling in the sense of Architecture Description Languages (ADL).

To better support reusable components, we split our system into two main parts:

1. Component-diagrams
2. Component-instance-diagrams

As typical for ADLs, this gives architects the opportunity of splitting a system into an architectural modeling layer (component-diagrams, where the basic system setup can be defined) and an instantiation layer (component-instance-diagrams, where components can be instantiated). This can be partly compared to the OMG Meta-Object-Facility (MOF, a four layer modeling architecture (OMG, 2015c)).

The following sections will introduce all necessary new language features.

3.1 Interface Definitions

A central role in our language is played by interfaces. GU-DSL supports standard interfaces (known e.g. from UML), but for our novel component extension, they are not sufficient. To make the system type- and communication-safe, we introduce two special keywords: *processor* and *provider*. As shown in Listing 1, the new keywords can directly be applied to either defining an output-interface (provider) or an input-interface (processor). The keywords classify the new interface types to be usable with component ports, as will be shown in Sec. 3.3.

```
Interface:
  visibility=('public'|'protected'|'
    private')? 'interface'
    ('provider'|'processor')? name=ID
    '{' (methods+=Method)* '}'
```

Listing 1: Component interface definitions.

An automatic communication specialization has been added in a way, that a provider-interface can talk to a corresponding processor-interface only. For this purpose, there is a restriction that a *provide*-method-signature needs a corresponding *process*-method-signature as can be seen in our example in Listing 8. It simplifies the usage, makes automatic code generation easier and allows for complete model validation. Which processor and which provider interface corresponds to each other can be defined in the component-diagram by connecting ports (see Sec. 3.3).

3.2 Component Definitions

Components and their simple (re-)usage are the central new contribution of our paper. In our approach, they are directly interconnected with the class-system known from UML class diagrams and are the base of our system architecture definition.

Therefore, an important novel idea is introduced with our class diagrams. The keywords ***source***, ***filter*** and ***sink*** can be used in a similar way as the standard *class*-keyword (see Listing 2). Stemming from the domain of data and image processing, these special keywords characterize the components as specified in Sec. 2.

Once having a component-class defined, it is used as basis for the final component. Designing components this way gives architects the possibility to split the definition (component-diagram) from the concrete implementation (class-diagram, activity-diagram). Components are designed in the newly introduced component-diagrams (Listing 3). As for class-diagrams, there are also the three main types (*source*, *filter*, *sink*) available. To support generic components that cannot be mapped to these three types, the *class*-keyword is usable. The component definition syntax (see Listing 3) itself is quite simple: it is the previously defined fully qualified name (Fqn) used in the class-diagram (see Listing 2). As class- and activity-diagrams, component-diagrams support nesting of additional components in arbitrary depth as can be seen in Listing 3.

```

ClassDiagram_:
'ClassDiagram' name=ID
  '{'
     classes+=BaseClass*
  '}';

BaseClass:
  visibility=('public'|'protected'|'private')?
    ('class'|'source'|'filter'|'sink') name=ID
    '{'
      (fields+=Field)*
      (methods+=Method)*
    '}'
```

Listing 2: Class-diagram and class definitions.

For embedded components, the same design principles are applicable as for all other components. That means the underlying classes have to be defined first and ports have to be used in the same way (see Sec. 3.3).

```
ComponentDiagram_:
  'ComponentDiagram' name=ID
  '{'
      (components+=Component)*
  '}';

Component:
  ('class'|'source'|'filter'|'sink'|  name=[BaseClass|
     Fqn]
  '{'
    (components+=Component)*
    (ports+=Port)*
    (connections+=Connection)*
  '}';
```

Listing 3: Component-diagram and component definitions.

3.2.1 Component Interfaces

Components require interfaces to be able to communicate. Hence, two types of component interface implementations are provided:

1. Interfaces by inheritance
2. Anonymous interfaces

Interfaces by Inheritance. are the default interfaces that should be used during the component definition. In this case, the component classes inherit from the necessary interfaces and provide all the methods that can later be used by ports and during the component functionality implementation. An important feature is that in this case, the interfaces methods have to be implemented only, if they require a special implementation. Otherwise, they are automatically available assuming a default implementation. This has the advantage of interface methods being directly accessible and usable without any additional code. However, if a special kind of implementation is required, the interface methods can also be implemented using our DSL.

Anonymous Interfaces. are a special version of interface implementations. In this case, the component class does not have to implement the interface, but the interface method can still be called directly, e.g. in the DSL's sequential expression language. Therefore we have added a new kind of interface call (Listing 4).

```
InterfaceCall:
  ('call' ('interface')? )? interfaceName=Fqn '.'
     methodName=ID
    '(' parameterValues+=ParameterValue
     (',' parameterValues+=ParameterValue)* ')'
```

Listing 4: Anonymous interface calls.

Anonymous interfaces are completely loose and the full interface definitions do not have to be available during design time. This gives designers the possibility to model components without having the full interface code. It can also be used to call interface methods from arbitrary places during component execution.

3.3 Port Definitions and Connections

Ports are the door to the outer and inner world of components. Allowing data exchange between components and nested components, ports provide the ability to define interfaces that handle how data can be transferred to and from a component. Therefore the previously defined interfaces (see Listing 1) can be chosen. The port definition within a component has a simple syntax (see Listing 5).

```
Port:
  ('async' ('buffered')? )? 'port' name=Fqn
    '(' connectors+=Connector
     (',' connectors+=Connector)* ')'

Connector:
  'processor'|'provider' name=Fqn ':' interfaceref=[
     Interface|Fqn];

Connection:
  '[' name=ID 'source' sourceport=[Connector|Fqn] '==>'
      'target' targetport=[Connector|Fqn] ']';
```

Listing 5: Port and Connection definitions.

The special preceded keyword *async* characterizes the port to be asynchronous, which means communication is forced to be non-blocking. Compared to standard blocking ports, a called interface method directly returns to the component and its sequential execution. A special variant of asynchronous ports are buffered ports. Incoming data can be buffered and is forwarded to the component if needed.

The detailed usage of components in combination with interfaces and ports can be found in our example in Sec. 4.

Ports play an important role when designing the system. By connecting two ports (source ==¿ target), we define which interfaces and which interface methods can be connected in the component-instance-diagram (see Sec. 3.4).

Once all components, interfaces, ports and connections are defined, the architecture definition is completed and the instantiation phase can begin as we will show in the next sections.

3.4 Component Instance Definitions

Once components have been designed and interface/-port wiring is completed, the architectural definition can be used in the component-instance-diagram. The component-instance-diagram is an important new feature we have added to be able to reuse components and to guarantee a type- and interface-safe component wiring. The diagram can be compared to the object-diagram known from UML, but it is specially designed for component based engineering. Splitting the instantiation from the definition of types and objects is a well-known practice in software engineering, but is not fully established for component based ADL approaches (or also UML) in this form.

The separation of definition and instantiation ensures two things:

1. Defined components can simply be reused and parametrized
2. Interface connections can be checked for validity during design

Using the component and port definitions of the component-diagram, the component instantiation can be realized as shown in Listing 6.

While the component definition just defines fields and methods in the class declaration, the instantiation step assigns a unique instance name and initialization values to a component and its fields. Three points are necessary to completely instantiate a component. The first one is a unique name. As second point, the component type has to be assigned using the fully qualified name of a component defined in the component-diagram. The third important point is the assignment of new initialization values. They can directly be assigned within two parentheses after the type specification by a name-value combination.

```
ComponentInstanceDiagram_:
  'ComponentInstanceDiagram' name=ID
  '{'
      compinstances+=ComponentInstance*
  '}';

ComponentInstance:
  name=ID 'instantiates' componentType=[BaseClass|Fqn]
      '(' (fieldName+=ID '=' fieldValue=FieldValue
          (',' fieldName+=ID '=' fieldValue=FieldValue)
      *)? ')'
  '{'
      (ports+=CID_Port)*
      ((bindings+=Binding)*)
      (compinst+=ComponentInstance*)?
  '}';
```

Listing 6: Component instance definitions.

Besides the component instantiation, the other important step is having connections allowing the communication and data-exchange between components. For this purpose, instances of ports are created first. Newly instantiated ports can then be wired. This syntax can be seen in Listing 7.

```
CID_Port:
    name=ID ':' type=ID
    '(' connectors+=CID_Connector (',' connectors+=
      CID_Connector)* ')' ';';

CID_Connector:
    'processor' | 'provider' name=ID '='
      processorInterfaceType=Fqn
    '(' (fieldName+=ID '=' fieldValue=FieldValue (','
      fieldName+=ID '=' fieldValue=FieldValue)*)? ')';

Binding:
    ('[' 'source' sourceport=[CID_Connector|Fqn] '==>'
      'target' targetport=[CID_Connector|Fqn] ']');
```

Listing 7: Component connection definitions.

Connections are possible between processor and provider ports only. This is guaranteed by the architecture definition specified in the component-diagram. Validity checks can be applied by an OCL (Object Constraint Language, see (OMG, 2015d)) model validator (used in our approach) or any other kind of validator. Depending on the final kind of code generation and how ports are implemented on the target platform, it is also possible to initialize and assign values to port instances. This can e.g. be necessary for buffered ports shown in Listing 5, giving the opportunity to define the buffer size and so forth.

The connections are created between instantiated ports. Only port instances are used as source or target. The direction is unidirectional and it does not matter if the source is a provider and the target a processor or vice versa.

3.5 Component Initialization and Execution

While the previous sections have shown how components can be defined in general, this section shows how they can be implemented. Hence, three methods are automatically available. The first one is responsible for initialization (***init()***) and it is assumed to be called during the instance initialization, while the second method is called during the component execution ***process()***. The third important method is the ***cleanup()*** and is responsible to release e.g. memory or other resources. It is important to know that we assume, that the ***process()***-method is executed in a thread. But this depends on the code-generator and the underlying CBSE framework. The implementation of all three methods is performed in the corresponding component class, either by using activity-diagrams or sequential code as we show in our example in Sec. 4.

3.6 Graphical Design Assistance

Besides the textual new features, we have also added special graphical editors allowing full graphical modeling in the sense of model driven development. Using graphical editors, system architects profit from the higher level of abstraction. Multiple lines of code can be added by drag and drop operations from a toolbox, placing e.g. complete components on a component-diagram. Also, connections can intuitively and interactively be added by simply selecting source and target ports. Sequential coding of method implementations is added using in-place editors supporting the full language specification, especially the expressions language. This gives architects the possibility to decide on their own, if they prefer either textual, graphical or mixed design. Since textual and graphical representations are fully compatible (round-trip engineering), the designer starts e.g. with textual modeling and can go over to graphical modeling and vice versa. This can facilitate the entry to this new kind of programming language and the component based engineering. An example of the graphical design system can be seen in Sec. 4.

3.7 Summary

The previous sections have introduced our novel textual and graphical language contributions as provider and processor interfaces, components, ports, component-instances and also component- and component-instance-diagrams. We have shown how components can be designed and how they can be connected using ports and interfaces.

Once the system architecture is specified in the class- and component-diagrams, component, port and connection instances have to be created in the novel component-instance-diagram. The separation of definition and instantiation in this case has the big advantage of giving two clear points of view onto the system implementation. Furthermore, the new component-instance-diagrams give the opportunity to uniquely name and initialize the component and port instances. By the usage of e.g. OCL validators, it is guaranteed that names and field-value assignments are unique and that instantiated component connections are applicable (provider to processor only).

All the previously introduced novel language features allow for a complete architectural system specification and instantiation. An example system can be found in Sec. 4.

4 A COMPONENT BASED MODELING EXAMPLE

In the next sections, we will show a small image processing example scenario to evaluate and demonstrate the simplicity and flexibility of our new modeling language (see also Fig. 2). A *Camera*-component (e.g. a color camera) acquires an image, which is mean-value filtered in the *MeanFilter*-component. At the end, the image is shown in the *Viewer*-component. We start with the definition of the necessary classes and interfaces in Listing 8.

Until now, we have defined the three main component classes and the corresponding image processor and image provider interfaces. The classes implement their required interfaces as shown in Sec. 3.2. Furthermore, they implement the required component *process*-method by calling the responsible activity diagrams (not part of this example). Listing 9 shows the usage of the classes as basis for components. The components themselves define all valid ports (*CameraImageProviderPort*, *MeanImageProcessorPort*, *MeanImageProviderPort*, *ViewerImageProcessorPort*) and connections (*CameraToFilter*, *FilterToViewer*) describing the final software architecture. As stated in Sec. 3, this means that we define which components are connectable. In this example, we allow the connection between a *Camera* and a processing *MeanFilter* and also the connection between a *MeanFilter* and a *Viewer*.

As can be seen, the basic architecture definition is quite simple. While Listing 9 shows the required component definitions and thus the developed architectural definition, Listing 10 instantiates the final system using a component-instance-diagram (*ComponentInstanceExampleDiagram*). It creates two Camera instances (*Camera1*, *Camera2*), a MeanFilter (*MeanFilter1*) and also a Viewer (*Viewer1*). Property values are also assigned during creation.

```
ClassDiagram ComponentClassDefinitions
{
  public interface provider IImageProvider
  { // see also the corresponding processImage
      signature
    public void provideImage(ref Image image);
  }

  public interface processor IImageProcessor
  { // see also the corresponding provideImage
      signature
    public void processImage(ref Image image);
  }

  public source Camera implements IImageProvider
  {
    public int width;
    public int height;

    public bool process(ConstIObjectParametersPtr
      parameters)
    { // Call the activity-diagram responsible for
      // image acquisition
      call behavior AdAcquireImage;
      return true;
    }
  }

  public filter MeanFilter implements IImageProvider,
      IImageProcessor
  {
    public bool process(ConstIObjectParametersPtr
      parameters)
    { // Call the mean filter activity-diagram
      call behavior AdFilterMean;
      return true;
    }
  }

  public filter Viewer implements IImageProcessor
  {
    public bool process(ConstIObjectParametersPtr
      parameters)
    { // Call the activity-diagram to show a new image
      call behavior AdShowImage;
      return true;
    }
  }
}
```

Listing 8: Component class example diagram.

Furthermore, all required port instances (*Camera1ProviderPort*, *Camera2ProviderPort*, *MeanProcessorPort*, *MeanProviderPort*, *ViewerProcessorPort*) are created. Finally, the ports are connected. As already mentioned, the instantiation of ports and also the connections are restricted to the fixed definitions in Listing 9.

As stated in Sec. 3.6, besides the textual DSL, we also propose graphical editors with this approach. It rounds off the new component system approach as can be seen in Fig. 3. The textual class-, component- and component-instance-diagrams are shown in their graphical version. While the textual form grows up relatively fast, the graphical representations are more compressed and much simpler to understand, which can simplify design.

The listings and figures in this section show a representative small example of the newly proposed DSL component features. Starting with the class- and interface-definitions (Listing 8), continuing with the component design (Listing 9) and finishing with the component instance definitions (Listing 10), the example creates a small image processing pipeline (Two cameras -¿ mean-filter -¿ viewer) in a textual and graphical form.

By using e.g. a code generator, the example can be transformed into compilable code. A reference implementation of a corresponding C++ CBSE system can be found in (Hoegg et al., 2015). [1]

5 RELATED WORK

This paper presents a novel DSL designed for textual and graphical, component based modeling. The next sections will have a look at some related work.

Several related languages and software engineering frameworks have been developed during the last years. Architecture Description Languages (ADLs) play an important role for this development, allowing description of hardware as well as software architectures. We exemplary compare our DSL and its infrastructure with the most important of them.

A very early approach of formal architecture definitions is *Wright* (Allen and Garlan, 1997). It was introduced by R. Allen et. al in 1997, proposing ways how components can be interconnected.

In (Magee et al., 1993), Magee et al. have introduced *Darwin*, a configuration language that allows for grouping process instances (an early kind of modern components) communicating by message passing.

Another standardized example of an ADL is *AADL* (Feiler et al., 2005). It was developed especially for automotive embedded hard- and software real-time systems, supporting several different types as devices, buses, processors (hardware-side) and threading or data processing (software-side). Furthermore, it can encapsulate visible functionality into connectable components.

UML (OMG, 2015e) and SysML (OMG, 2015f) are standardized, ADL like languages supporting structural, sequential and behavioral graphical modeling of real world problems. They are proposed to be generic applicable system and architecture languages, giving architects the possibility of graphically designing and solving problem descriptions.

Besides ADLs, also other related component based approaches and frameworks have been installed in several domains of software engineering in the past two decades.

An early example of a component based robotic controlling framework is SmartSoft (Schlegel and Wrz, 1999) on basis of CORBA (Common Object Request Broker Architecture (OMG, 2015a)).

[1]Examples and additional material: http://www.gu-dsl.org

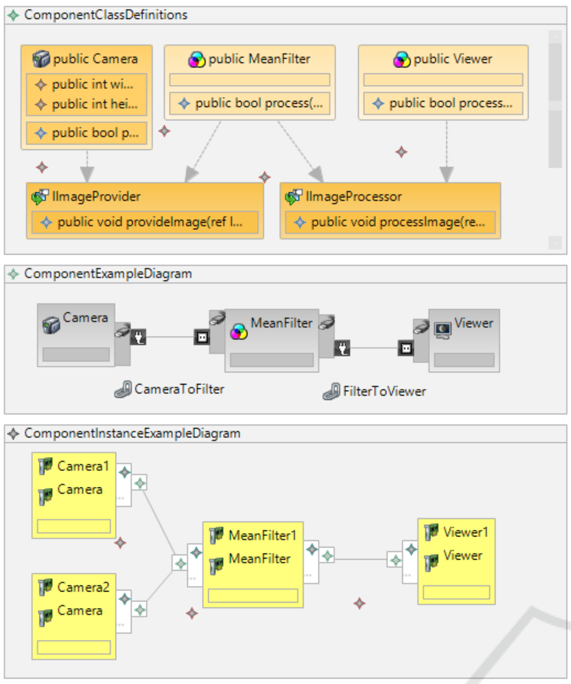

Figure 3: The graphical CBSE example. Top: The class diagram; Middle: The component-diagram; Bottom: The component-instance-diagram.

```
ComponentDiagram ComponentExampleDiagram
{
  source ComponentClassDefinitions . Camera
  { // Define the output port
    port CameraImageProviderPort ( provider
      CameraImageProviderPort :
              ComponentClassDefinitions .
      IImageProvider );
    // Define the port
    [CameraToFilter source CameraImageProviderPort .
      CameraImageProviderPort =>
          target MeanImageProcessorPort .
      FilterImageProcessorPort ]
  }

  filter ComponentClassDefinitions . MeanFilter
  { // Define the input and output ports
    port MeanImageProcessorPort ( processor
      FilterImageProcessorPort :
              ComponentClassDefinitions .
      IImageProcessor );
    port MeanImageProviderPort ( provider
      FilterImageProviderPort :
              ComponentClassDefinitions .
      IImageProvider );

    // Define the port
    [FilterToViewer source MeanImageProviderPort .
      FilterImageProviderPort =>
          target ViewerImageProcessorPort .
      ViewerImageProcessorPort ]
  }

  sink ComponentClassDefinitions . Viewer
  { // Define the input port
    port ViewerImageProcessorPort ( processor
      ViewerImageProcessorPort :
                ComponentClassDefinitions .
      IImageProcessor );
  }
}
```

Listing 9: Component example diagram.

```

ComponentInstanceDiagram
      ComponentInstanceExampleDiagram
{
  Camera1 instantiates ComponentClassDefinitions .
      Camera
      (width = 640, height = 480)
  {
    Camera1ProviderPort : CameraImageProviderPort(
      provider CameraImageProviderPort =
      IImageProvider());

    [ source Camera1ProviderPort .
      CameraImageProviderPort => target
      MeanFilter1 . MeanProcessorPort .
      MeanImageProcessorPort ]
  }

  Camera2 instantiates ComponentClassDefinitions .
      Camera()
  {
    Camera2ProviderPort : CameraImageProviderPort(
    provider CameraImageProviderPort = IImageProvider
      ());

    [source Camera2ProviderPort .
      CameraImageProviderPort => target
     MeanFilter1 . MeanProcessorPort .
      MeanImageProcessorPort ]
  }

  MeanFilter1 instantiates ComponentClassDefinitions .
      MeanFilter()
  {
    MeanProcessorPort : MeanImageProcessorPort(
    provider MeanImageProcessorPort = IImageProcessor
      ());

    MeanProviderPort : MeanImageProviderPort(
    provider MeanImageProviderPort = IImageProcessor
      ());

    [source MeanProviderPort . MeanImageProviderPort =>
      target
    Viewer1 . ViewerProcessorPort .
      ViewerImageProcessorPort ]
  }

  Viewer1 instantiates ComponentClassDefinitions .
      Viewer()
  {
    ViewerProcessorPort : ViewerImageProcessorPort(
    provider ViewerImageProcessorPort =
      IImageProvider());
  }
}
```

Listing 10: Component instance example diagram.

They propose concepts and patterns how electronic and software components of a robot can interact and how they can be designed in a reusable way. Furthermore, they introduce concepts of event driven communication and also model driven approaches (Schlegel et al., 2009).

In 2002, the CORBA Component Model (CCM) (OMG, 2015b), based on CORBA 3.0, has been released supporting component based distributed architectures and services.

Another widely spread approach of component based architectures is Microsoft's Component Object Model (COM) and its distributed version (DCOM) (Microsoft, 2015). It is mostly language independent using instantiable interfaces for an abstract view of distributed components.

(Hoegg et al., 2015) introduce an image processing C++ CBSE system. Using a code generator, GUDSL can be translated into this kind of component framework.

In contrast to the mentioned languages and component systems, GU-DSL and its infrastructure provides the full functionality necessary for developing interactive image processing pipelines. Furthermore, the DSL supports the concept of round-trip engineering necessary for arbitrary switching between graphical and textual modeling. It specially focuses on object oriented textual and graphical modeling in the sense of model driven engineering. Starting with object abstractions (from object to class to component) up to the final system generation in the future, the full development process can be assisted.

Besides the presented ADLs, frameworks and concepts, many other systems exist, e.g. LabView (Instruments, 2015) with its dataflow visual programming language. But as far as we know, no language exists which is comparable to the proposed one in the domain of data and image processing.

6 CONCLUSION

In this paper we presented GU-DSL, a component based textual and graphical language. GU-DSL uses some well known design principles as classes, interfaces and components. These principles are adapted and improved to fit into the domain of data and image processing. We discussed all the novel features and concepts (provider and processor interfaces, components, ports, component-instances and also component- and component-instance-diagrams) allowing the architecture definition and instantiation. Furthermore, we have demonstrated their usage in an image processing example.

A big advantage of the proposed work, compared to other concepts (such as UML), is the combination between graphical and textual modeling and the supported round-trip engineering. This means that both kinds of modeling are always synchronized in both directions. Starting with textual modeling does not prevent developers from switching over to graphical modeling and also back. So it is up to the modeler's and programmer's preferences which kind of technique is used. Experienced programmers can decide for textual programming, enjoying the full feature set necessary for component-based engineering with the advantage of much better guidance and support than generic programming languages provide.

For the future, we plan to extend the graphical editors by a better and more complete toolbox (required for graphical object creation). Also, the usability has to be improved. Furthermore, the support of operating systems other than Windows is planned for the language infrastructure and a code generator for C++ CBSE system has to be developed.

ACKNOWLEDGEMENTS

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

REFERENCES

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249.

Efftinge, S., Eysholdt, M., Khnlein, J., Zarnekow, S., and Contributors. (2015). Xtext 2.5 documentation.

Feiler, P. H., Lewis, B., Vestal, S., and Colbert, E. (2005). An overview of the sae architecture analysis & design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering. In *Architecture Description Languages*, volume 176, pages 3–15. Springer US.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Hoegg, T., Koehler, C., and Kolb, A. (2015). Component based data and image processing systems - a conceptual and practical approach. In *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on*, pages 66–69.

Instruments, N. (2015). Ni labview. Available at http://www.ni.com/labview/.

Magee, J., Dulay, N., and Kramer, J. (1993). Structuring parallel and distributed programs. *Software Engineering Journal*, 8:73–82.

Microsoft (2015). Component object model (com). Available at https://msdn.microsoft.com.

OMG (2015a). Corba. Available at http://www.corba.org/.

OMG (2015b). Corba component model. Available at http://www.omg.org/spec/CCM.

OMG (2015c). Metaobject facility (mof). Available at http://www.omg.org/mof/.

OMG (2015d). Ocl. Available at http://www.omg.org/spec/OCL/.

OMG (2015e). Uml. Available at http://www.uml.org/.

OMG (2015f). Uml. Available at http://www.omgsysml.org/.

Schlegel, C., Hassler, T., Lotz, A., and Steck, A. (2009). Robotic software systems: From code-driven to model-driven designs. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–8.

Schlegel, C. and Wrz, R. (1999). Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework smartsoft. In *Advanced Mobile Robots, 1999. (Eurobot '99) 1999 Third European Workshop on*, pages 195–202.