A component concept for scientific experiments – focused on versatile visual component assembling

Kerstin Falkowski
Institute for Software Technology, University of Koblenz-Landau
Universitätsstrasse 1, 56070 Koblenz, Germany
falke@uni-koblenz.de

Abstract

The intent of this PhD thesis is the specification, prototypical implementation and evaluation of a component concept for scientific experiments, focused on versatile visual component assembling by a human user, assisted by a comprehensive assembling environment. This is achieved by providing a visual assembling simultaneous from different viewpoints.

1 Introduction

In the field of computer science most research areas comprise a *huge amount of data structures, algorithms and characteristic algorithm chains* providing a basis for the research area's tasks. These are normally used and combined consistently in different ways to solve new problems.

In established research areas there are standard libraries implementing those basic elements efficiently, best practices for their use possibly becoming manifest in patterns and/or even standard tools providing ready-made behaviour for all basic tasks of the research area. In relatively young research areas such standards do not exist, which unfortunately often leads to re-developments of basic data structures, algorithms and algorithm chains on the green field again and again.

The component concept targeted in this work provides a platform superseding this repeated re-development of algorithms and especially of algorithm chains. It provides rules for developing algorithms in a specific way, so that they constitute *components*. Built thereon it provides a comprehensive assembling environment, in which a human user can *assemble components visually to a more complex component* (corresponding to an algorithm chain). Of course, assembled components can in turn be part of even more complex components (hierarchic assembling).

An example for a relatively new research area is *image processing*. Here, two totally different kinds of tools are used for research. On the one hand there are open source C/C++ APIs, like the *Open Computer Vision Library (OpenCV)*[1], implementing basic data structures and algorithms in an efficient manner, but rarely more complex algorithm chains. On the other hand there is the commercial standard software *MATLAB Image Processing Toolbox (IPT)*[2], also providing basic data structures and algorithms, that can be combined to more complex algorithms. This is done textually in some editor in a closed system. A lot of image processing researchers test their more complex algorithms in the MATLAB IPT before they implement them in C++ using some image processing API.

Using the planned component concept for scientific experiments, those researchers could implement basic image processing algorithms once as components and consistently use them for experiments via assembling them visually to arbitrarily complex components. In contrast to existing visual assembling environments, the assembling can be performed in a more versatile manner via a simultaneous assembling from different viewpoints.

The targeted component concept will be specified and implemented in a generic way, but evaluated using a large amount of well-defined image processing data structures and algorithms. Subsets of these data structures and algorithms are used as examples in the following.

This paper is structured as follows. Section 2 presents existing component concepts with interesting aspects for the planned component concept. Section 3 provides a detailed description of the targeted component concept and points out important research tasks. Section 4 summarises the current state of the PhD thesis and describes future work including implementation and evaluation.

2 Component concepts

This section provides a short introduction to existing component concepts with interesting aspects for the targeted component concept. There are a lot of existing sci-

[1]`http://opencv.willowgarage.com`
[2]`http://www.mathworks.com/products/image`

entific and industrial component concepts and even more definitions of the term component. In line with the literature [15, 10] we define a *component* as a reusable software unit conforming to a *component concept* with a *well-defined interface*, encapsulating its realisation and possible internal states. The interface forms a *contract* between the component and its environment. It specifies which services a component is able to provide, if all demands of the component including required services are met by the environment (e.g. by other components).

Component assembling is the activity that distinguishes the component lifecycle from the common software lifecycle. It can be carried out from different behavioural *viewpoints*. A *component concept* has different purposes. At first it has to determine how components can be assembled and especially how they can be composed. This includes the component interface, the kind of assembling environment, the assembler and the viewpoints for assembling. Built thereon it has to specify the resultant component lifecycle in detail, including its activities, their possible order(s) and their corresponding actors. Based on this conceptual part, realisation techniques can be chosen and a specific environment for component development, assembling and/or execution can be developed. Finally, all this in conjunction determines the capabilities of the component concept's components.

There are two well known surveys of existing software component concepts. Lau and Wang [10] analyse 13 different scientific and industrial component concepts, compare them based on three different aspects (component semantics, component syntax and component composition) and classify them into four different categories according to component composition. *The Common Component Modeling Example (CoCoME)* [13] was a challenge, where 13 developers mapped a given component-based architecture example to their own scientific component concept. Goal of the challenge was making various component concepts comparable in a way. As a result the participating component concepts were classified into four categories, in this case according to their focus. In addition Selmat et al. [14] compare technical aspects of different scientific and industrial component concepts, namely their interaction mechanism and support for distributed applications.

A component concept for scientific experiments should provide the visual assembling of components that are already implemented and supplied (and potentially occur as binaries) from different viewpoints. For that purpose, there are two interesting groups of existing component concepts.

The first group contains component concepts, that allow the visual assembling of already implemented and supplied components, which Lau and Wang classify in the category 'Deployment with Repository'. Examples are *JavaBeans* [9] combined with a suitable assembling environment like *JBeanStudio*[3] and *ConQAT* [6]. In both component concepts functionality is implemented in atomic components in terms of Java classes with specific properties, that are specified, developed and supplied at development time. These can be visually assembled to a composite component and directly executed in a specific assembling and runtime environment. In JavaBeans the composition "glue code" is stored in an adapter Java class, in ConQAT the composition information is saved as composed component in an own XML dialect (.cqb). In contrast to JavaBeans, ConQAT supports hierarchical component composition. Unfortunately both component concepts compose components only from one fixed viewpoint, JavaBeans from an *event-driven* viewpoint and ConQAT from a *data-flow* viewpoint.

The next interesting group contains component concepts, that model component-based architectures of complete applications from different viewpoints at design time, which form the CoCoMe category 'formal models, focusing on behaviour and quality properties'. Examples are *SOFA 2.0* [5], *Fractal* [4] and *Palladio* [3]. In all three component concepts a component is defined equivalent to the definition mentioned above. In SOFA and Fractal structural information about a component-based architecture is modelled conform to an *Ecore*[4]-metamodel and the behaviour of components is described by (extended) *Behaviour Protocols (BP)*. In Palladio four different views to a component-based architecture can be modelled, conforming to an *Ecore*-metamodel, and additionally performance relevant aspects for each of these models can be specified via *Service Effect Specifications (SEFFs)*. SOFA 2 and Fractal explicitly separate functional and control parts of components. All three provide hierarchical component composition. In Fractal and Palladio components are composed via interfaces bindings, in SOFA 2 via connectors offering different communication styles. Fractal provides shared components, component instances that are is subcomponents of several composite components. SOFA 2 and Palladio provide performance prediction at design time. Fractal provides the verification of composition correctness as well as of correspondences between code and behaviour specification. All three component concepts provide modelling tools, SOFA and Fractal additionally provide APIs for supporting the (top down) implementation of designed architectures. However, all three component concepts are originally modelling approaches for complete component-based architectures and do not provide the visual assembling of already implemented and supplied components.

[3]`http://www.vislab.usyd.edu.au/moinwiki/JBeanStudio`

[4]`http://www.eclipse.org/modeling/emf`

3 A component concept for scientific experiments

In this section the targeted component concept for scientific experiments is described in detail. As mentioned before, it offers the *visual assembling of components by a human user, assisted by an assembling environment*. In comparison to the other mentioned visual assembling environments in Section 2, the adaptation and especially the composition of components shall be much more versatile. This is achieved by providing *assembling from different viewpoints*, namely data-flow, service-usage and control-flow.

This leads to three different kinds of component composition (Section 3.4). On the one hand two components can be composed, if a datum of a component is required by the other component. This component composition constitutes a *data-flow*, similar to component composition in ConQAT. On the other hand, two components can be composed, if a service of a component is required by the other component. This component composition constitutes a *service-usage*, conforming to the common view on component composition, e.g. in UML 2.2 components. Moreover a component or one of its services can be an arbitrarily complex control structure. A component composition with such "control components" constitutes a *control-flow*.

Accordingly, a *component* may require various *input data* from its environment to perform its task and it can provide distinct *output data*. Moreover a component may require *services of other components* to perform its task and can provide at least one *own service* to other components. So, the *interface of a component* has to offer different kinds of *ports*[5] for receiving input data and service access from the environment and sending output data and access to its own services to the environment. The mentioned control components can be used in the internal structure of a component but they do not have an explicit impact on the interface of a component.

3.1 Component interface

A component comprises any number of input ports. *Data input ports* facilitate the receiving of input data from the environment. *Service input ports* enable the access to services of other components. Every input port has a *multiplicity* $m..n$ with $m, n \in \mathbb{N}$ and $n > 0$, that decides how much inputs from different sources the port can receive at runtime. Input ports whose lower bound m is 0 are called *optional input ports*, if m is greater than 0 the input ports are called *obligatory input ports*. Input ports whose upper bound n is 1 are called *single input ports*, if n is greater than 1 the input ports are called *multi input ports*.

If a datum/service is obligatory for a component to perform its task, its interface offers an obligatory service port, if the datum/service is optional, the interface offers an optional service port. Usually a component interface offers an optional data input port for a parameter, for which the component has a default value/object, that can but does not have to be changed by a component assembler. If two or more input data/services of the same type have a specific role for a component, its interface offers several single input ports. If the data/services have no specific role, the interface offers a multi input port.

An input datum can be passed directly to a component via component adaptation (Section 3.3). Moreover, input data/services can be sent/offered to a component by another component at runtime, where the source component is determined during component composition (Section 3.4).

Moreover, a component interface may comprise any number of *data output ports*, each of them facilitating the sending of exactly one output datum to the environment during runtime. One can distinguish between two different kinds of output data and thus between two distinct kinds of data output ports: If an output datum is a constant of a component, its interface offers a *constant data output port*, that is able to send the output datum anytime during runtime. If an output datum is a result of the component's task, its interface offers a *resultant data output port*, that is not able to send the output datum to the environment until the component performed its task.

A component interface comprises one *service output port* for every service of the component, offering access to this service to other components during runtime. A service output port is not able to offer access to its service until the component received all input data and/or services obligatory for the service.

Output data/services can be sent/offered by a component to any number of other components at runtime, where the target components are determined during component composition (Section 3.4).

Every port has a *defined type* that is decided during component development by a component developer and changed nevermore. All data ports beside constant data ports have in addition a *current type*. After component instantiation this is the same as the defined type, but during component assembling it can be changed to any subtype.

There can be dependencies between data and thus data ports of one component, affecting its adaptation and/or composition from a data-flow viewpoint. As an example the current type of an data input or output port can depend on the current types of one or more other data input ports. Such dependencies should explicitly be offered by the component's interface to its environment in a formalised form, so that an assembling environment as well as a component assembler can make use of them. A research task in this context is the specification of a suitable concept for the formalisation and persistent storage of those dependencies at

[5]Here port in terms of entry and exit is meant, not port in terms of UML.

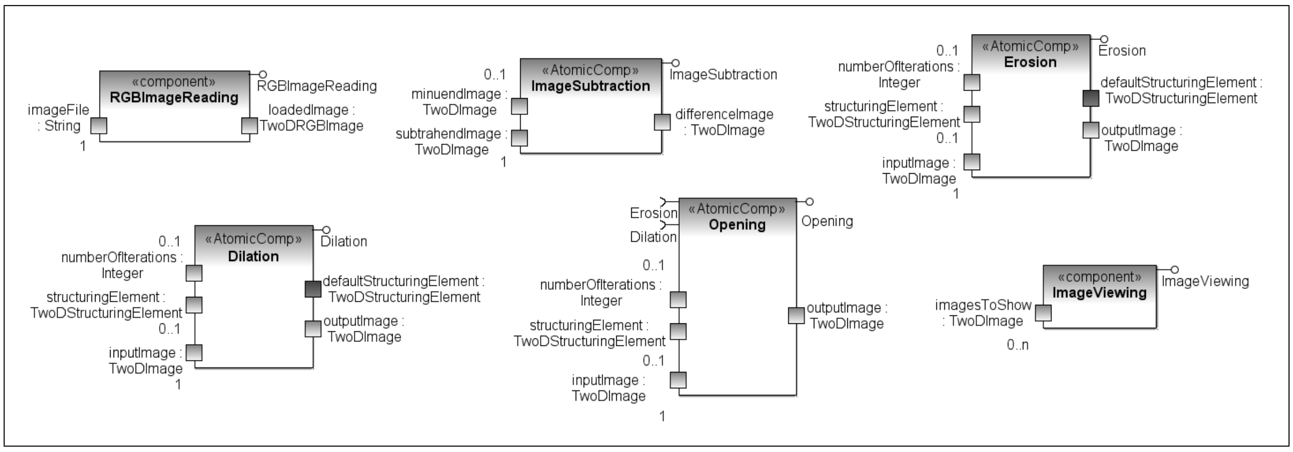

Figure 1. Basic components.

development time and their usage at assembling time and runtime.

A component interface shall be able to describe a component and its services as accurately as possible. This information can help an assembling environment to disallow a syntactically wrong assembling, and it can help a component assembler to potentially even avoid a semantically wrong assembling. Moreover detailed information can help to decide, if a component is equivalent to another component and can be substituted by it. So, the assembling environment shall be able to forward as much information about the component as possible to the component assembler. Orth [11] gives a good overview about distinct information for component description.

Example. In the following, some characteristic image processing operations are shortly introduced and conceptually modelled as exemplary components conform to the described component interface. The components are modelled via the use of UML component / composite structure diagrams in a specific way.[6]

An component offers its own services via UML provided interfaces in ball-notation on its right upper corner. A component that requires access to the services of other components possesses UML required interfaces with the names of the services in socket-notation on its left upper corner. For the visualisation of component composition from a data-flow viewpoint, UML ports are modified to data ports, rather functioning like UML pins. Input data ports are always displayed on the left side of a component, data output ports are always displayed on its right side. Constant data output ports are dark grey, in contrast to all other data ports, that are grey.

Figure 1 shows some components, as a base for component composition. `RGBImageReading` is a component, that loads an image from a file. It requires a String containing path and name of the image as obligatory input datum and delivers it as 2d RGB image as resultant output datum. `ImageSubtraction` is a component that subtracts one image element-wise from another image. It requires two 2d images of the same type and size as obligatory input data and delivers a 2d image of the same type and size as resultant output datum. To deliver an image of the same type, the image values of the resultant `differenceImage` have to be normalised. `ImageViewing` is a component that visualises arbitrarily images. It can get any number of images as input data and delivers no output data. `Erosion` takes for every output image value the minimum of neighbouring values of the input image. `Dilation` performs a contrary operation using the maximum of neighbouring values. The concrete neighbouring values are in both cases determined by a structuring element. `Opening` performs an erosion to an image followed by a dilation. All three components require a 2d image as obligatory input datum and deliver a 2d image of the same type as resultant output datum. Optionally they can get the number of iterations in terms of an integer $n > 0$ and a 2d structuring element as optional input data. `Erosion` and `Dilation` provide their default structuring elements as constant output data.[7] `Opening` additionally requires the services `Erosion` and `Dilation` from its environment.

3.2 Component instantiation

A component assembler has to instantiate a component, before it can be assembled. The assembling environment should facilitate a *visual component instantiation* via drag & drop from a list of existent components to an assembling canvas.

After it was instantiated a component has a state, that can be changed by component adaptation. A is the decision how

[6]Using the IBM Rational Software Architect `http://www.ibm.com/software/awdtools/architect/swarchitect`.

[7]It is a design decision of the component developer, that the opening operator has no default structuring element as constant output datum.

instantiated components and their adapted values/objects are stored, directly via serialising component instances or indirectly in specific external files.

3.3 Component adaptation

Via *visual component adaptation* a component assembler can directly pass input data to instantiated components.

For that purpose the assembling environment should provide specific *adaptation GUIs*. Such an adaptation GUI has to present the current assembling state of a component, including already adapted and/or composed input data, and it has to facilitate the change of input data values regarding potentially existent dependencies between the component's data. Thereby it has to *disallow an incorrect adaptation*, e.g. setting a value of a wrong type or outside the correct range to an input datum and so forth.

An adaptation GUI has to be generated by the assembling environment on demand based on a component interface. To enable the automatic generation of adaptation GUIs, there have to be *predefined GUI elements* for all input data types of a component. These GUI elements can be offered by the component concept as part of the assembling environment or developed by a component developer. The component concept should provide GUI elements for all atomic types as well as for often used complex standard types, like e.g. `File`. A component developer should be able to add GUI elements for any type that is used by a developed component.

One important research question in this context is, if an additional visual component adaptation during runtime shall be enabled. A further research task is the specification of a suitable concept for the development and integration of type-specific GUI elements at development time and their automatic composition to adaptation GUIs at assembling time and runtime. Here related work concerning GUI builders, like the *Eclipse Visual Editor project*[8] or parts of the JavaBeans API, partly examined in [11], has to be regarded.

3.4 Component composition

Via *visual component composition* a component assembler can determine, which instantiated components send/offer what data/services to which other instantiated components at runtime.

The assembling environment should facilitate a component composition via drawing an arrow from an output port of a component to a corresponding input port of another component. Thereby, it is responsible for *disallowing an incorrect composition*, e.g. the composition of two ports of the same component, the composition of two input ports or two output ports, the composition of a data port and a service port, the composition of two ports with incompatible types and/or a cyclic data-flow composition of two or more components and so forth.

A component `A` can use another component `B` in two different ways. `A` can use `B` *directly* via receiving one or more of its resultant data after `B` performed its task. Or `A` can use `B` *indirectly* via getting access to one or more of `B`s services after `B` received all required input data/services. These two kinds of usage should not be combined, because for an indirect use `B` has to receive potentially less input data/services than for a direct use. For that purpose an assembling environment could prohibit a simultaneous usage in both ways. This means, if at least one of the resultant output ports of a component is composed, its service output ports can not be composed and vice versa. They can be disabled or even made invisible.

There is a further research task concerning component composition in general, that affects realisation of the component concept but is not only a realisation problem. It has to be decided, if data/services are passed from one component to another during execution by value/copy or by reference. Or rather, in which cases data/services are sent by value/copy and in which cases by reference. And, who determines, if data are sent by reference or by value/copy: the component developer, the component assembler and/or the assembling environment.

Data-flow. During component assembling components can be *composed from a data-flow viewpoint* via their *data ports* (Section 3.1). Such a data flow composition means, that at runtime one component sends a specific datum to another component.

Every data output port of a source component can be connected to any number of data input ports of target components, if their types are *compatible*. For a compatibility checking, the data port's *current type* is used and can be changed due to composition. Only for a constant data output port, the defined type is used, because it has no current type. Usually one would expect, that an output port is compatible to an input port, if the output port type is *the same or a subtype* of the input port type. This statement is correct, but can be *incomplete*. As mentioned before, there can be dependencies between the ports of one component, and these dependencies can lead to further cases of compatibility via changing the current types of depending ports.

Hence, further research tasks are the analysis of the impact of port dependencies on a data-flow component composition and the development of an algorithm that checks the compatibility of two data ports and determines which current types have to be changed in which way to offer the data port composition. Here, related work dealing with *type inference* [12] has to be regarded.

Service-usage. During component assembling components can be *composed from a service-usage viewpoint* via

[8]`http://www.eclipse.org/vep/`

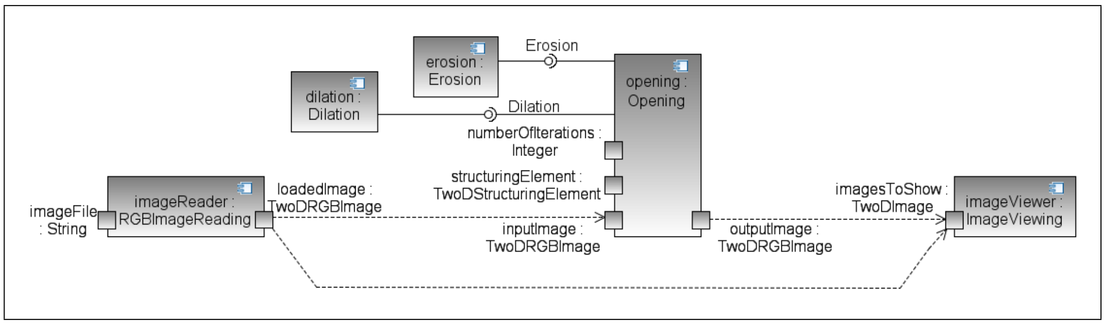
erosion : Erosion
Erosion
dilation : Dilation
Dilation
opening : Opening
numberOfIterations : Integer
structuringElement : TwoDStructuringElement
imageFile : String
imageReader : RGBImageReading
loadedImage : TwoDRGBImage
inputImage : TwoDRGBImage
outputImage : TwoDRGBImage
imagesToShow : TwoDImage
imageViewer : ImageViewing

Figure 2. Component composition.

their *service ports* (Section 3.1). Such a service-usage composition means, that at runtime a component's service is used by another component.

Every service output port of a source component can be connected to any number of service input ports of target components, if their types are *compatible*, which means that the output port type is *either the same or a subtype* of the input port type.

Usually one would expect, that a component can offer its services anytime during runtime. This is because, a target component uses a service internally and is itself responsible for providing all required input data and/or services to the service before use. But this statement is only true, if the two service ports have exactly the same type. If a service output port type is a subtype of the service input port type, this subtype can specify additional obligatory input data and/or services that the target component does not know. In this case, the source component is not able to provide the service to the target component, until it received those additional obligatory input data and services.

Control-flow. During component assembling, components can also be assembled from a *control-flow viewpoint*, because *every component is itself a control structure*.

Every component affects the successive program flow during runtime *implicitly* in different ways. Usually a component assembler does not really know, how the adaptation and/or composition of a component correlates to the resulting program flow at runtime. But if this knowledge is made explicit during assembling time, it can facilitate an *explicit* control-flow component composition.

On the one hand, an assembling environment can provide *ready-made generic control components* for different kinds of loops and choices. So, a research task is to identify important control structures and develop them in a generic way, conforming to the component concept. Here, existing *visual languages dealing with control flow* like UML activity diagrams [1] and languages for exogenous connectors like Reo [2], where atomic 'channel types' can be hierarchically composed to more complex connectors, have to be regarded.

On the other hand, a component developer can develop *arbitrarily complex domain specific control components* for his research area. Hence, a research task is to analyse the impact of such control structures to the program flow and summarise the results in terms of guidelines for domain specific control component development.

A control component decides, how often the services of its composed components are used or rather how often its composed component are executed.

Example. Figure 2 shows a set of assembled components, not (yet) saved as composed components. A component composition from a data-flow viewpoint is visualised as a connection between an output port of a component and an input port of another component. A component composition from a service-usage viewpoint is visualised as an assembly connection in lollipop notation. Because the `RGBImageReader` delivers a 2d RGB image, the input image port and output image port of the `Opening` component changed their types from `TwoDImage` to the subtype `TwoDRGBImage`.

3.5 Component storage

A component assembler can *store a set of assembled components persistently as a new component.*

A set of assembled components is *storable*, if all components are able to either perform their task or offer their services at runtime. This is the case, if at storage time none of them possesses an obligatory input port, that is neither adapted nor composed. If there are still unassembled obligatory input ports, they can be added to the interface of the new component. Then the required data and/or services have to be provided from the environment of the new component at runtime and are internally passed to the contained components.

The assembling environment should facilitate the storage of a component via one click. Thereby it is responsible for *disallowing the storage of incomplete components*. This can be achieved by automatic addition of unassembled obligatory input ports to the new component interface.

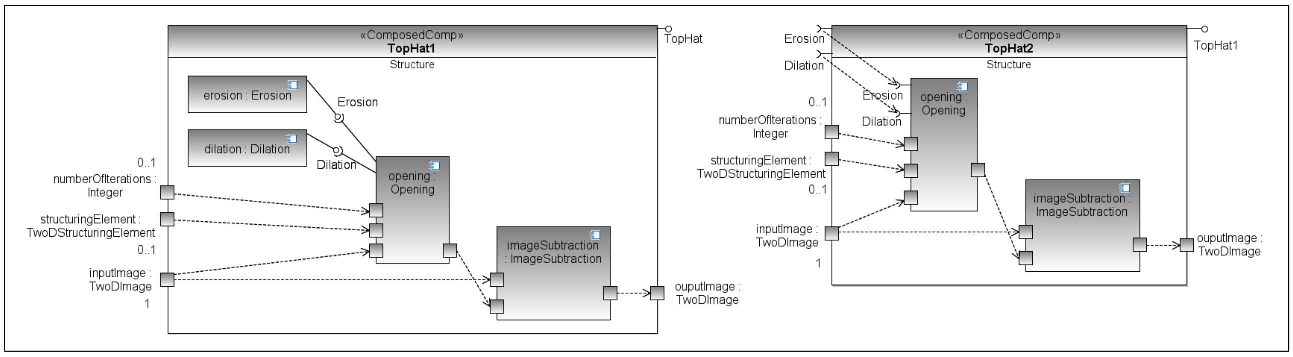

Figure 3. Top hat components.

Every stored component can again be loaded into the assembling environment for further (re-)assembling or instantiated to be used in another component. Moreover it can be executed, if its interface comprises no unassembled obligatory input ports (Section 3.6).

Example. Figure 3 shows two different component variants of the morphological operation `TopHat`, that subtracts the opening of an image from the original image. Both top hat operators compose the components `ImageSubtraction` and `Opening` internally. `TopHat1` delivers the services `Erosion` and `Dilation` to `Opening` internally via the components `Erosion` and `Dilation`. `TopHat2` delivers the services `Erosion` and `Dilation` to `Opening` via requiring them from the environment itself. Both top hat components require a 2d image as obligatory input datum and deliver a 2d image of the same type as resultant output datum. Optionally they can get the number of iterations in terms of an integer $n > 0$ and a 2d structuring element as optional input data. In both components the dependencies between their data input and output ports can be derived from the internal modelling.

3.6 Component execution

A component user can initiate the execution of an executable component by the assembling environment. A component is *executable*, if its interface comprises no unassembled obligatory input ports. For execution it has to be loaded into the assembling environment.

The assembling environment controls the execution and is responsible for a correct performance. It carries out the following evaluation algorithm:

1. Pass all received input data to contained components.

2. Send constant output data from all contained components to connected contained components.

3.a Execute all contained components, that are able to perform their task now, and send resultant data to connected contained components.

3.b Offer service access from all contained components, that are able to offer their service now, to connected contained components.

4. Repeat step 3 until there are no further components that can be executed or can offer their service.

5. Send resultant data and offer services to connected components.

3.7 Component concept realisation

The component concept will be realised using *Java* as well as *TGraphs* [7], directed graphs whose vertices and edges are typed, ordered and attributed, and *JGraLab*[9], an API for TGraph processing.

During component development the component concept explicitly distinguishes between two kinds of components. An *atomic component* is developed by a component developer at development time. It can be supplied as .java and/or .class file, packed in a .jar archive, depending on how much information the component assembler/user shall get about the atomic component's internals. A *composed component* is developed by a component assembler during assembling time in the assembling environment. Internally it is modelled as TGraph instance conforming to a specific TGraph schema for composed components. It can be persistently stored as .tg file, packed in a .jar archive, and again loaded from this .tg file. But both kinds of components have *exactly the same interface* to their environment during visual component assembling. The *defined type* of a data port can be *any Java type*. The *defined type* of a service port has to be a subtype of *a specific Java type* constituting the supertype for all components.

An atomic component is instantiated via instantiation of the corresponding Java class and a composed component is instantiated via loading the corresponding TGraph into Java and instantiating all contained components. An atomic component is executed via calling the `execute()`-method of its corresponding Java class. A composed component is executed by executing directly used contained components.

For the development of adaptation GUIs and adaptation GUI elements potentially JavaBeans *property editors*

[9]`jgralab.uni-koblenz.de`

and/or customisers [9] can be used. For the formalisation of dependencies between data ports, potentially the *Java Modeling Language*[10] can be used.

The mentioned image processing components for evaluating the component concept shall use existing image processing libraries. Haas [8] gives a good overview about those libraries.

4 Summary

As mentioned at the beginning, the intent of the PhD thesis is the specification, prototypical implementation and evaluation of a component concept for scientific experiments. In section 3 a detailed description of the targeted component concept and predicted research tasks have been introduced. The research tasks comprise:

1. a) The specification of a suitable concept for the formalisation and persistent storage of dependencies between data ports of one component at development time and their usage at assembling time and runtime. b) The analysis of the impact of those port dependencies on a data-flow component composition. c) The development of an algorithm that checks the compatibility of two data ports and determines which current types have to be changed in which way to offer the data port composition.

2. a) The identification of important control structures and their development as ready-made generic control components conforming to the component concept. b) The analysis of the impact of arbitrarily complex domain specific control structures on the program flow and the summarisation of the results in terms of guidelines for the development of such control components.

3. The specification of rules that determine, in which cases data/services are passed from one component to another during execution by value/copy and/or by reference, and/or rules that determine, which actor takes this decision.

4. a) The determination, how instantiated components are stored. b) The decision, if components can additionally be adapted at runtime. c) The specification of a suitable concept for the development and integration of type-specific GUI elements at development time and their automatic composition to adaptation GUIs at assembling and runtime.

In future work, these research tasks have to be performed to complete the specification for the planned component concept for scientific experiments. Based thereon the component concept can be prototypically implemented and evaluated using a large amount of image processing data structures and components[11].

[10] `http://www.eecs.ucf.edu/~leavens/JML`

[11] The planned component concept is developed in the context of the project *Software Techniques for Object Recognition (STOR)*, `http://er.uni-koblenz.de`, whose task is the use of software engineering techniques in image processing.

References

[1] UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.

[2] F. Arbab. *Reo: a channel-based coordination model for component composition*, volume 14. Cambridge University Press, New York, NY, USA, 2004.

[3] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, 2009.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[5] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[6] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[7] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In R. Gimnich, U. Kaiser, J. Quante, and A. Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, 2008. GI.

[8] J. Haas. Analyse, Evaluation und Vergleich von Bildverarbeitungsbibliotheken aus Sicht der Softwaretechnik. Master's thesis, Universität Koblenz-Landau, 4 2009.

[9] G. Hamilton (Editor). JavaBeans. Technical report, Sun Microsystems, 08 1997. Version 1.01-A.

[10] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 10 2007.

[11] S. Orth. Entwicklung eines Konzepts zur Selbstauskunftsfähigkeit für STOR-Komponenten. Master's thesis, Universität Koblenz-Landau, 12 2009.

[12] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, New York, NY, USA, 1994. ACM.

[13] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil. *The Common Component Modeling Example: Comparing Software Component Models*. Springer Publishing Company, Incorporated, 2008.

[14] M. H. Selamat, H. Sanatnama, R. Atan, and A. A. Abd Ghani. Software Component Models from a Technical perspective. *International Journal of Computer Science and Network Security (IJCSNS)*, 7(10):135 – 147, 10 2007.

[15] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2002. (first edition 1997).