

A Generic Component Framework for System Modeling

Hartmut Ehrig¹, Fernando Orejas²,
Benjamin Braatz¹, Markus Klein¹, and Martti Piirainen¹

¹ Technische Universität Berlin,
Franklinstrasse 28/29, 10587 Berlin, Germany
{ehrig,bbraatz,klein,martti}@cs.tu-berlin.de

² Universidad Politècnica de Catalunya,
Campus Nord, Mòdul C6, Jordi Girona 1-3, 08034 Barcelona, Spain
orejas@lsi.upc.es

Abstract. The aim of this paper is to present a generic component framework for system modeling which is especially useful for a large class of graph- and net-based modeling techniques. Moreover, the framework is also flexible with respect to a hierarchical connection of components, providing a compositional semantics of components. This means more precisely that the semantics and internal correctness of a system can be inferred from the semantics of its components. In contrast to constructor-based component concepts for data type specification techniques, our component framework is based on a generic notion of transformations. Refinements and transformations are used to express intradependencies, between the export interface and the body of a component, and interdependencies, between the import and the export interfaces of different components. This is shown by a small case study on modeling Java threads by high-level Petri nets in this paper.

1 Introduction

It is becoming a standard practice in software development to base the design and implementation of component-based systems on architectures such as CORBA or COM+. In these architectures, components are generic in the sense that they are, to some extent, independent of specific programming languages. Unfortunately, components in these frameworks lack a precise semantics, making difficult to reason about this kind of systems. This is probably due to the fact that these approaches are mainly addressed to system implementation, but do not include the modeling phase.

1.1 Main Concepts and Results of this Paper

The aim of this paper is to present a generic component framework for system modeling that can be used for a large class of graph- and net-based modeling

techniques, but in principle also for other kinds of semi-formal and formal techniques. More precisely, we present in Sect. 2 a component concept based on a very general notion of specification. According to this concept, a component consists of a body and of an import and an export interface, connected in a suitable way, such that the import connection defines an inclusion from the import interface to the body, and the export connection defines a suitable transformation from the export interface to the body. These import and export connections represent the intradependencies between different parts of a single component. The interdependencies between import and export of different components are represented by connectors. Again, we only require connectors to define a suitable transformation. Consequently, our framework is also generic concerning the connection of components. The key concept of our framework is a generic notion of transformations of specifications, especially motivated by - but not limited to - rule based transformations in the sense of graph transformation and high-level replacement systems [8, 5, 19].

According to the general requirement that components are self contained units not only on the syntactical but also on the semantical level, we are able to define the semantics of each component independently of other components in the system. This semantics is also given in terms of transformations. It must be pointed out that this semantics can be used to give meaning to components based not only on a formal, but also on a semi-formal modeling technique. In this paper, however, we mainly consider graph- and net-based techniques. Moreover, our transformation semantics is shown to be compositional. More precisely, we are able to show that the semantics of a system can be inferred from that of its components.

In order to illustrate our concepts, we present in Sect. 3 a small case study where Java threads ([11]) are modeled by high-level Petri nets. A larger case study concerning a component-based telephone service center modeled by low-level Petri nets is given in [18], where the corresponding component concept is an instantiation of our generic framework. This and other kinds of instantiation are discussed in Sect. 4. A short summary and open problems are presented in Sect. 5.

1.2 Related Work

The generic component concept, presented in this paper, has been mainly motivated by the ideas in [15] for a component concept in the German BMBF project “Continuous Software Engineering” and by the module concepts for graph transformation systems in [20] and for Petri nets in [18]. In contrast to these concepts and that for UML [3] an important new aspect in our framework is the fact that we are able to give a self-contained, compositional semantics for each component. The syntactical level of the approaches in [20, 18] is partly motivated by algebraic module specifications in [6]. The semantics of algebraic module specifications is based on free constructions between import and body part. This constructor-based semantics has been dropped in [18, 20] for graph- and net-based modules, where the key concepts are now refinements between export and body parts.

This leads directly to our transformation-based component framework, where transformations include refinements and abstractions.

Although we mainly focus on our new component framework in this paper, we think that the concepts introduced here are a good basis for other interesting architectural issues. In this sense, we think that the papers [9, 23, 24, 1] could be considered complementary to ours. In particular, the use of graph transformation techniques proposed by Fiadeiro in [23] and also by Löwe in [13] for architecture reconfiguration seems to be most promising, in view of a component concept for continuous software engineering, including software reconfiguration and evolutionary software development in the sense of [15].

2 Main Concepts for a Generic Component Framework

In this section, we present the main concepts for our generic component framework. We start with some general assumptions concerning our modeling technique, which is one of the key generic concepts in our framework.

2.1 Generic Modeling Techniques

We propose that a generic modeling technique is a general framework for describing systems. These descriptions have a syntactical part, consisting of specifications *SPEC*, and a semantical part, consisting of behaviour or models of the corresponding specification. Moreover, it should be possible to deal with specific formal or semi-formal modeling approaches as concrete instances of the generic technique. A modeling technique is called formal, if the syntactical and semantical parts are mathematically well-defined. For a semi-formal technique we only require that the syntactical part is formalized, but the semantical part may only be given by informal documents, like natural language.

In order to express properties of behaviours or of models, we assume that we have a constraint language, which allows us to formulate constraints in an informal way, using diagrams or natural language, or in a formal way, based on some logical formalism. For simplicity we do not consider constraints explicitly in this paper.

From the software engineering point of view, we also require to have suitable horizontal and vertical structuring techniques. Especially in this paper we require a suitable notion of transformation including abstraction and refinement of specifications as special cases.

2.2 A Generic Component Concept

Components are self-contained modeling units, where some details are hidden to the external user. This is usually achieved by providing a clear separation between the interface of the component (what the external user, or other components, can “see”) and the body (the detailed definition or implementation of the functionality provided by the component). Moreover, the interface can be

divided into two parts: the import interface, describing what the component assumes about the environment (e.g. a description of the services provided by other components) and the export interface, describing the services provided by the component itself. Obviously, the import and export interfaces are connected to the body in some well-defined way.

In this sense, given a generic modeling technique with model specifications in the sense of 2.1, we are now able to define our *generic component concept*. A *component specification*, in short *component*,

$$COMP = (IMP, EXP, BOD, imp, exp)$$

consists of model specifications and connections:

- IMP , called *import interface*,
- EXP , called *export interface*,
- BOD , called *body*,
- $imp: IMP \rightarrow BOD$, called *import connection*,
- $exp: EXP \rightarrow BOD$, called *export connection*.

In order to be generic, we do not require any specific type of connections between interfaces and body. We only require that each export connection, $exp: EXP \rightarrow BOD$, uniquely defines a transformation of model specifications (see 2.3), $exp: EXP \Rightarrow BOD$, called *export transformation*. We assume that this transformation is a *refinement* describing how the elements presented in the export interface are implemented by the body. In other words the export is an *abstraction* of the body.

With respect to the import connection, we may assume that the body of a component is an extension of the import interface, in the sense that the functionality defined in the body is built upon the elements of the import interface. As a consequence, for the sake of simplicity, we assume that each import connection, $imp: IMP \rightarrow BOD$, defines an inclusion $imp: IMP \subseteq BOD$, of the corresponding specifications.

2.3 A Generic Transformation Concept

We need a generic transformation concept in order to formulate properties of export connections (see 2.2) and of connectors between import and export interfaces of different components (see 2.5, below). Again, we will try to be as general as possible.

We assume that a transformation framework \mathcal{T} consists of a class of transformations, which includes identical transformations, is closed under composition and satisfies the following *extension property*: For each transformation $trafo: SPEC_1 \Rightarrow SPEC_2$, and each inclusion $i_1: SPEC_1 \subseteq SPEC'_1$ there is a selected transformation $trafo': SPEC'_1 \Rightarrow SPEC'_2$, with inclusion $i_2: SPEC_2 \subseteq SPEC'_2$, called the *extension* of $trafo$ with respect to i_1 , leading to the extension diagram in Fig. 1.

$$\begin{array}{ccc}
SPEC_1 & \xrightarrow{trafo} & SPEC_2 \\
\downarrow i_1 & & \downarrow i_2 \\
SPEC'_1 & \xrightarrow{trafo'} & SPEC'_2
\end{array}$$

Fig. 1. Extension diagram for the extension property

It must be pointed out that, in a given framework, given *trafo* and i_1 as above, there may be several *trafo'* and i_2 , that could satisfy this extension property. However, our assumption means that, in the given framework \mathcal{T} only one such *trafo'* and one inclusion i_2 are chosen, in some well-defined way, as the extension of *trafo* with respect to i_1 .

The idea underlying this extension property is to ask a transformation framework to satisfy, what we may call, a locality assumption: if one can apply a transformation on a certain specification, then it should be possible to apply the “same” transformation on a larger specification. This assumption has been formulated, in a more precise way in [16]. In this paper, for the sake of simplicity, we have avoided the technical details.

We could have also required that these extensions would only exist when the given *trafo* is consistent with i_1 in a specific sense. For instance, in the case of graph transformations, the extension property corresponds to the embedding of a transformation into a larger context. The corresponding embedding theorem in [8] requires that the “boundary” of i_1 has to be preserved by *trafo*. Again, for the sake of simplicity, in this paper we drop this consistency condition.

If transformations and inclusions of specifications can be considered as suitable morphisms in a category of specifications, then the extension diagram may be a pushout in this category. In general, however, the extension diagram is not required to be a pushout. Especially the embedding of graph transformations discussed above does not lead to a pushout in general.

2.4 Transformation Semantics of Components

According to general requirements, components should be self-contained units, with respect to syntax and semantics. Hence, it is necessary to have a semantics for each single component. Moreover, the semantics of composite components (and, eventually, the entire system) must be inferred from that of single components. In this subsection, we propose a semantics of components satisfying these requirements.

The most standard way of defining the semantics of a given component concept consists in considering that the meaning of a component is some kind of function mapping models of the import to models of the export interface like in the case of algebraic module specifications ([6]). Unfortunately, this kind of semantics is not adequate for our purposes in this paper. There are two main problems. On one hand, this kind of semantics assumes that the specifications involved in a component have a well-defined model semantics. This is true in the

case of a formal modeling technique, but not in the case of a semi-formal one. On the other hand, this kind of semantics implicitly assumes that the import interface of a component is a loose specification having more than one model (otherwise, if the import has just one model the function associated to a component becomes trivial). However, this is not the case for components used in connection with most graph- and net-based modeling techniques. For instance, as can be seen in our small case study, the import interface of a Petri net component is a Petri net, which one would typically consider to define a single model.

The solution provided for these problems is a semantics that takes into account the environment of a component, in a similar way as the continuation semantics of a programming language assigns the meaning of a program statement in terms of the environment of the statement. Here, the idea is to think that, what characterizes the import interface of a component is not its class of models, but the possible refinements or transformations of this interface that we can find in the environment of the component. In this sense, it is natural to consider that the semantical effect of a component is the combination of each possible import transformation, $trafo: IMP \Rightarrow SPEC$, with the export transformation $exp: EXP \Rightarrow BOD$ of the component. Since IMP is included in BOD , we have to extend the import transformation from IMP to BOD in order to be able to compose both transformations. Due to the extension property for transformations, we obtain $trafo': BOD \Rightarrow SPEC'$, as shown in Fig. 2.

$$\begin{array}{ccccc}
 EXP & \xrightarrow{exp} & BOD & \xrightarrow{trafo'} & SPEC' \\
 & & \uparrow imp & & \uparrow imp' \\
 & & IMP & \xrightarrow{trafo} & SPEC
 \end{array}$$

Fig. 2. Transformation semantics

Let us call the class of all transformations $trafo: IMP \Rightarrow SPEC$ from IMP to some specification $SPEC$ the *transformation semantics* of IMP , denoted by $Trafo(IMP)$, and similar for EXP . According to Fig. 2 the *transformation semantics* of the component $COMP$ can be considered as a function

$$TrafoSem(COMP): Trafo(IMP) \rightarrow Trafo(EXP)$$

defined for all $trafo \in Trafo(IMP)$, by $TrafoSem(COMP)(trafo) = trafo' \circ exp \in Trafo(EXP)$.

It may be pointed out that the two problems mentioned above are now solved. On one hand, semiformal modeling techniques may include precise notions of transformation or refinements (the case of visual modeling techniques is briefly discussed in Sect. 4.4). Similarly, a Petri net may have just one model, but it may be refined in many different ways. Therefore, in both cases, there would not be a problem to define a transformation semantics for these kinds of components.

2.5 Composition of Components

Several different operations on components can be considered in our generic framework. Depending on the underlying architecture intended for a given system, some specific operations may be needed. For instance, in a hierarchical system one could need an operation to compose components by matching the import of one (or more) components with the export of other component(s). On the contrary, on a non-hierarchical system one may need an operation of circular composition, where the components of a system are connected in a non-hierarchical way. Especially connectors in the sense of [1] and the architectural description language WRIGHT should be considered in this context.

In the following, for the sake of simplicity, we only consider one basic operation, which allows one to compose components $COMP_1$ and $COMP_2$ by providing a *connector*, $connect: IMP_1 \rightarrow EXP_2$, from the import interface IMP_1 of $COMP_1$ to the export interface EXP_2 of $COMP_2$. Similar to an export connection, we only require that the connector uniquely defines a transformation $connect: IMP_1 \Rightarrow EXP_2$.

Different generalisations and variations of this operation, for instance by allowing to compose (simultaneously) several components, would not pose too many difficulties, only some additional technical complication. Circular composition may be more difficult to handle, however previous experience in dealing with module operations (see e.g. [6]) would provide good guidelines.

Now, we are able to define the composition

$$COMP_3 = COMP_1 \circ_{connect} COMP_2$$

as follows. Let $xconnect = exp_2 \circ connect$. The extension property implies a unique extension $xconnect': BOD_1 \Rightarrow BOD_3$, with inclusion $imp_1': BOD_2 \subseteq BOD_3$ in Fig. 3. The composition $COMP_3$ is now defined by

$$COMP_3 = (IMP_3, EXP_3, BOD_3, imp_3, exp_3)$$

with $imp_3 = imp_1' \circ imp_2$ and $exp_3 = xconnect' \circ exp_1$. Since we have $IMP_3 = IMP_2$ and $EXP_3 = EXP_1$, this means especially that the result of the composition concerning the interfaces is independent of the body parts.

Note, that each connector $connect: IMP_1 \rightarrow EXP_2$ can also be considered as a separate component $COMP_{12}$ with $exp_{12} = connect$ and $imp_{12} = id_{EXP_2}$. This allows to consider $COMP_3$ in Fig. 3 as the composition of three components $COMP_1$, $COMP_{12}$ and $COMP_2$, where all connectors are identities.

2.6 Compositionality of Transformation Semantics

Given a connector, $connect: IMP_1 \rightarrow EXP_2$, between IMP_1 of $COMP_1$ and EXP_2 of $COMP_2$, the composition $COMP_3$ of these components via $connect$ is well-defined, and we have the following compositionality result:

$$\begin{aligned} TrafoSem(COMP_3) = \\ TrafoSem(COMP_1) \circ Trafo(connect) \circ TrafoSem(COMP_2) \end{aligned}$$

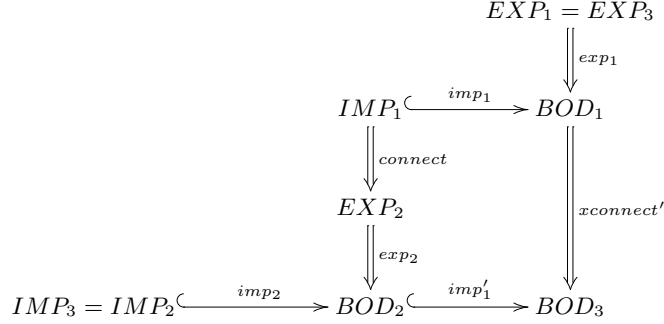


Fig. 3. Composition of Components

where $Trafo(connect)(trafo) = trafo \circ connect$. This means that the transformation semantics of the composition $COMP_3$ can be obtained by functional composition of the transformation semantics of $COMP_1$ and $COMP_2$ with a most simple intermediate function

$$Trafo(connect): Trafo(EXP_2) \rightarrow Trafo(IMP_1)$$

defined above.

In order to prove this important compositionality result, we only need to require that the extension property for transformations is closed under horizontal and vertical composition. This essentially means that the horizontal and vertical composition of extension diagrams, as given in Fig. 2, is again an extension diagram. This is very similar – and in some instantiations equal – to the well-known horizontal and vertical composition of pushouts in Category Theory.

3 Modeling Java Threads with Components in the Framework of High-Level Nets

As a small case study we will model a few aspects of the behaviour of threads in the programming language Java with algebraic high-level Petri nets. For a larger case study we refer to [18]. An overview reference for Java threads is given in [11].

We use a notation where an algebraic high-level net N consists of an algebraic signature $\Sigma_N = (S, OP, X)$ with sorts S , operation symbols OP and variables X , a Σ_N -algebra A_N and a Petri net, where each place pl has a type $type(pl) \in S$, the in- and outgoing arcs of a transition tr are inscribed with multisets of terms over Σ_N and a transition tr itself is inscribed with a set of equations over Σ_N . In our example, however, we only use single terms as arc inscriptions and do not use equations for transitions.

We use a transformation concept similar to [17] based on rules and double pushouts in a suitable category of high-level nets. In our example we directly present the corresponding high-level net transformations. The extension property

is satisfied because the redex of a rule applied to a net is preserved by the inclusion into another net. The application of the rule to the larger net yields the extended transformation.

Our small case study consists of two components $COMP_1$ and $COMP_2$ leading to a composition $COMP_3$ as shown in Fig. 3.

3.1 Implementation of the run-Method

In the first component $COMP_1$, we define a rough model of the lifecycle of a thread in the export, and refine it in the body by adding a *run*-transition that represents the execution of the thread. This step corresponds to the extension of the `Thread`-class by a class implementing a *run*-method.

The export signature Σ_{EXP_1} consists of one sort *Thread* and two constant symbols $thread_1: \rightarrow Thread$ and $thread_2: \rightarrow Thread$ representing two different threads of control. The net structure of the export interface is shown in Fig. 4. The type of all places is *Thread* and all arcs are inscribed with the variable t of sort *Thread*.

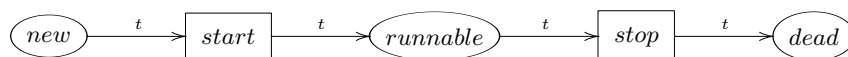


Fig. 4. EXP_1

The export interface corresponds to the fact that the class `Thread` has a *start*-method which makes a newly created thread runnable and a *stop*-method which kills a runnable thread:

```

public class Thread {
    public void start() { ... }
    public void stop() { ... }
    ...
}
  
```

The export transformation $exp_1: EXP_1 \Rightarrow BOD_1$ refines the signature by adding a sort *RunState* representing the states that can occur during the execution of *run*, a sort *Object* representing the states an arbitrary object can be in, operations $st: ThreadObject \rightarrow RunState$ and $con: \rightarrow Object$ and an operation $do: Object \rightarrow Object$ representing the run-time changes to the object. The net structure is refined by removing the *stop*-transition and adding places *started* and *finished* of type *RunState* and transitions *run* and *exit*. The details can be found in Fig. 5. We have replaced the *stop*-transition by an *exit*-transition because now the thread can exit normally by completing its task. This is modeled by the *run*-transition that transfers the *RunState* from *started* to *finished*.

The addition of the *run*-transition corresponds to the implementation of the *run*-method and the *Object* in the constructor *st* of *RunState* to the existence of an attribute object that is changed by the *run*-method:

```

class MyThread extends Thread {
  
```

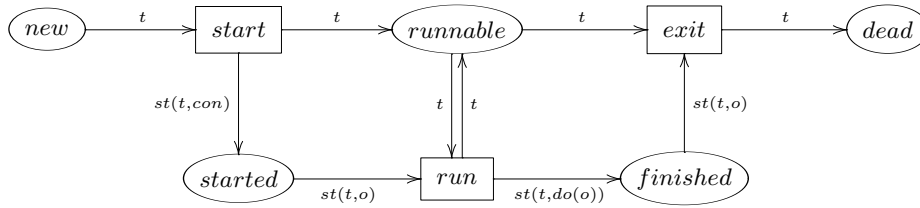


Fig. 5. BOD_1

```

Object anObject;
public void run() { ... }
}

```

In the import interface (Fig. 6) only the *run*-transition and the adjacent places are kept, because this transition is useful to be further refined.

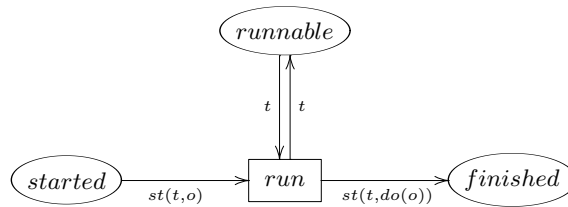


Fig. 6. $IMP_1 = EXP_2$

3.2 Further Refinement of the Method

In the second component $COMP_2$ the *run*-transition is refined by a model with two phases. The export interface EXP_2 is the same net as the import interface IMP_1 of $COMP_1$. The export transformation $exp_2: EXP_2 \Rightarrow BOD_2$ adds two new operations $do_1, do_2: Object \rightarrow Object$ to the signature. The *run*-transition is removed and replaced by two new transitions *act1* and *act2* with an intermediate place *working* (see Fig. 7). We assume that the algebra A_{BOD_2} satisfies the equation $do_2(do_1(o)) = do(o)$ because sequential firing of *act1* and *act2* should still produce the same result as before.

This replacement corresponds to a further extension of `MyThread`, where the `run`-method is overwritten with a method that does the same by calling two sequential actions:

```

class MyThread2 extends MyThread {
    public void run() {
        act1();
        act2();
    }
    private void act1() { ... }
}

```

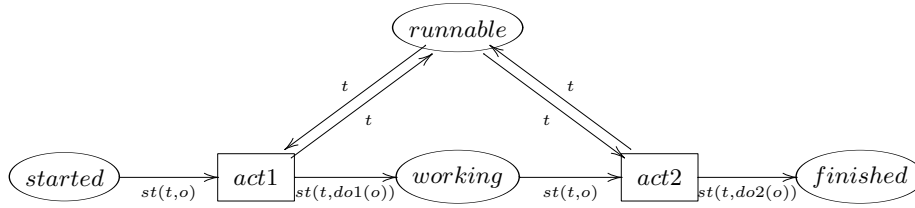


Fig. 7. BOD_2

```
private void act2() { ... }
}
```

The import interface IMP_2 could consist of the whole body, if both transitions should be refined further, but to make it more interesting, we assume that $act1$ is already an atomic action and only $act2$ shall be refined. This leads to an import (Fig. 8) with only the transition $act2$ and the adjacent places.

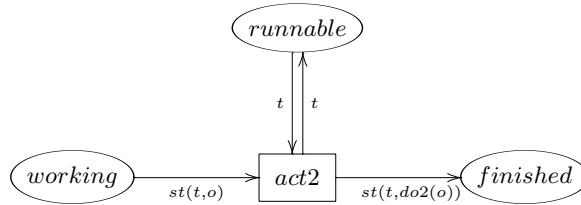


Fig. 8. IMP_2

3.3 Composition of the Components

The composition of the two components $COMP_1$ and $COMP_2$ presented above with identical connection from IMP_1 to EXP_2 yields a component $COMP_3$ with $EXP_3 = EXP_1$, $IMP_3 = IMP_2$ and a body BOD_3 (Fig. 9) resulting from application of the rule underlying exp_2 to the net BOD_1 , replacing the transition run by $act1$ and $act2$.

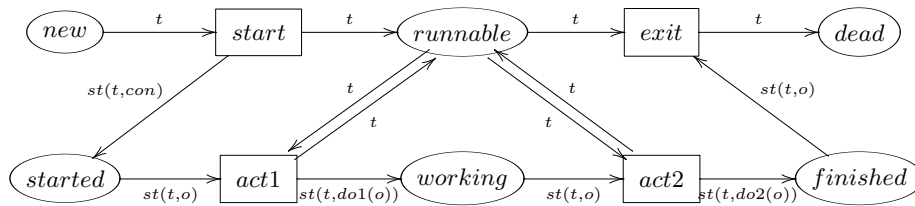


Fig. 9. BOD_3

4 Instantiations of the Generic Component Framework

We have pointed out, already, that our component framework is generic with respect to several aspects. First of all, it is generic with respect to the technique used for system modeling. Secondly, it is generic with respect to the semantics of components and with respect to the composition operation, using a generic notion of connector, where the semantics and the connectors are based on a generic notion of transformation.

In this section, we discuss instantiations of the generic component framework leading to existing and new component concepts for different kinds of Petri nets and graph transformations, as well as some ideas for other visual modeling techniques, like UML.

4.1 Instantiations of Transformations

Transformation and refinement concepts have been developed for a great variety of data type and process modeling techniques in the literature. In the following, we will focus on algebraic graph transformation concepts and transformation of high-level structures [8, 5]. The extension property for transformations formulated in Sect. 2 is well-known, as an embedding theorem, in the case of algebraic graph transformations based on the double pushout approach. This approach has been generalized to the categorical framework of high-level structures, including low- and high-level Petri nets. Several concepts and results for algebraic graph transformations have been generalized to the categorical framework of high-level structures and replacement systems, including a basic version of the embedding theorem. The horizontal and vertical composition property, required in 2.6, can be shown under suitable assumptions for high-level replacement systems and instantiated for Petri nets and other modeling techniques. Explicit examples of transformations in the case of algebraic high-level nets are given in our small case study in Sect. 3.

4.2 Component Concepts for Low- and High-Level Petri Nets

Various kinds of Petri nets are suitable for our framework. Rule-based transformations of Petri nets, in the sense of graph transformations and high-level replacement systems, can be used as instantiation of transformations in the generic component concept of Sect. 2. In fact, two different kinds of Petri nets have been considered for the formal instantiation: place/transition nets and algebraic high-level nets. But, also, other kinds of low-level and high-level nets are suitable. A specific example of components, in the case of algebraic high-level nets, has been given in Sect. 3, including composition of components. This component concept for high-level Petri nets is closely related to the component concepts for low-level Petri nets presented in [18]. In fact, refinements of transitions, in the sense of [18], are closely related to rule-based transformations of Petri nets, where the left-hand side of the rule consists of a single transition only. This means that the larger case study in [18], in the application domain of telephone services, is another example of our component concept.

4.3 Component Concepts for Graph Transformation Systems

Similar to the case of Petri nets discussed above, also different kinds of graph transformation systems are suitable for our framework. M. Simeoni ([20]) has considered a notion of refinement for graph transformation systems, which is used as export connection between the graph transformation systems in the export and in the body part. It should be pointed out that a refinement of graph transformation systems is conceptually different from a refinement or transformation of graphs. Actually, the structures to be refined or transformed in the first case are graph transformation systems, i.e., a set of graph productions, while the structures are graphs in the second case. In fact the refinements of graph transformation systems, in the sense of [20], can be considered as transformations in the sense of 2.3, because they are shown to satisfy the extension property. More precisely, it is shown in [20] that the corresponding category of refinements has pushouts, if at least one morphism is an inclusion. This property is shown for typed algebraic graph transformation systems ([10]) and local action systems in the sense of [12]. In both cases this means that the extension diagram is a special kind of pushout considered in [20]. This means that these two module concepts for graph transformation systems with interesting examples in [20] can be considered as different instantiations and examples of our generic component concept. In [7] we discussed already an extension towards attributed graph transformations ([14]) which leads now to a new component concept for this case. It has to be checked how far other module concepts for different kinds of graph transformation systems fit into our framework.

4.4 Towards a Component Concept for Visual Modeling Techniques

As shown by R. Bardohl [2], a large number of visual modeling techniques can be formulated in the framework of GENGED, based on attributed graph transformations ([14]). It is shown how attributed graph transformations can be used to define transformations of visual sentences for different visual modeling techniques. This seems to be a good basis to define a component concept for visual modeling techniques as instantiations of the general framework presented in this paper. This applies especially to the different diagram formalisms of the UML ([22]), where already suitable simplified versions of class diagrams and statecharts have been considered in the framework of GENGED.

More directly, there are already suitable refinement and transformation concepts for different kinds of UML techniques, which might be used as transformations in the sense of our generic component concept. However, it remains open which of these transformation concepts satisfy our extension property in order to obtain other interesting instantiations of our component concept. In particular, in [4], a transformation concept for statecharts has been introduced, which can be considered as an instantiation of transformations for high-level replacement systems, as discussed in 4.1. This seems to be a good basis for a component concept for statecharts.

5 Conclusion

We have presented a generic component framework for system modeling. This framework is generic in the sense that it can be instantiated by different system modeling techniques, especially by graph- and net-based integrated data type and process techniques. Moreover, it is based on a generic transformation concept, which is used to express intra- and interdependencies of interfaces and bodies of components.

Our proposed framework meets to a large extent the requirements stated in [15] for a component concept for continuous software engineering. Due to lack of space, in this paper we just study a simple operation of (hierarchical) composition of components. However, we have already studied a union operation for components and our experience on algebraic specification modules provides good guidelines to define other more complex operations like, for instance, circular composition. Moreover, it remains open to extend our framework to architectural issues in the sense of [9, 23, 13, 24, 1] discussed in the introduction and Sect. 2.5.

We have defined components not only as syntactical, but also as self contained semantical units. This allows to obtain the important result that our semantics based on transformations is compositional.

In our paper [7] we have given a conceptual and formal framework for the integration of data type and process modeling techniques. In a forthcoming paper we will show how this framework can be combined with our component approach in this paper leading to a component framework for a generic integrated modeling technique with special focus on different kinds of constraints.

Finally let us point out again that our generic framework is suitable not only for formal graph- and net-based modeling techniques mainly discussed in this paper, but also for other formal and semi-formal techniques. However, it remains open to tailor our framework to specific requirements for visual modeling techniques including UML and to relate it to existing capsulation, package and component concepts for some of these techniques ([3]).

Acknowledgements: This work is partially supported by the German DFG project IOSIP within the DFG priority program “Integration of Software Specification Techniques for Applications in Engineering”, the German BMBF project on “Continuous Software Engineering” and by the Spanish project HEMOSS (TIC 98-0949-C02-01). We would like to thank the FASE '02 referees for several valuable comments leading to a much more comprehensive paper.

References

1. R. Allen, D. Garlan. A Formal Basis for Architectural Connection. In *ACM TOSEM '97*, pp. 213–249.
2. R. Bardohl. GENGED – *Visual Definition of Visual Languages Based on Algebraic Graph Transformation*. PhD Thesis, TU Berlin, Verlag Dr. Kovac, Germany (1999).
3. J. Cheesman, J. Daniels. *UML Components*. Addison-Wesley (2001).

4. H. Ehrig, R. Geisler, M. Klar, J. Padberg. Horizontal and Vertical Structuring for Statecharts. In *Proc. CONCUR '97*, Springer LNCS 1301 (1997), pp. 327–343.
5. H. Ehrig, A. Habel, H.-J. Kreowski, F. Parisi-Presicce. Parallelism and Concurrency in High-Level Replacement Systems. In *Math. Struct. in Comp. Science 1*. Cambridge Univ. Press (1991), pp. 361–404.
6. H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, vol. 21 of *EATCS Monographs on Theor. Comp. Science*. Springer Verlag, Berlin (1990).
7. H. Ehrig and F. Orejas. A Conceptual and Formal Framework for the Integration of Data Type and Process Modeling Techniques. In *Proc. GT-VMT 2001, ICALP 2001 Satellite Workshop*. Heraclion, Greece (2001), pp. 201–228. Also in *Electronic Notes in Theor. Comp. Science 50,3* (2001).
8. H. Ehrig, M. Pfender, H. Schneider. Graph Grammars: An Algebraic Approach. In *Proc. SWAT '73*, pp.167 - 180.
9. J.L. Fiadero, A. Lopes. Semantics of Architectural Connectors. In *Proc. TAPSOFT '97, Springer LNCS 1214* (1997), pp. 505–519.
10. R. Heckel, A. Corradini, H. Ehrig, M. Löwe. Horizontal and Vertical Structuring of Typed Graph Transformation Systems. In *MSCS, vol. 6* (1996), pp. 613–648.
11. Cay S. Horstmann, Gary Cornell. *Core Java 2. Volume II – Advanced Features*. Sun Microsystems Press, Prentice Hall PTR (2000).
12. D. Janssens, N. Verlinden. A Framework for ESM and NLC: Local Action Systems. In *Springer LNCS 1764* (2000), pp. 194–214.
13. M. Löwe. Evolution Patterns. In *Proc. Conf. on Systemics, Cybernetics and Informatics 1999, vol. II*, pp. 110–117.
14. M. Löwe, M. Korff, A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In *Term Graph Rewriting: Theory and Practice* (1993), chapter 14, pp. 185–199.
15. S. Mann, B. Borusan, H. Ehrig, M. Große-Rhode, R. Mackenthun, A. Sünbül, H. Weber. Towards a Component Concept for Continuous Software Engineering. FhG-ISST Report 55/00 (2000).
16. F. Orejas, H. Ehrig, E. Pino. Tight and Loose Semantics for Transformation Systems. To appear in *Proc. Workshop on Algebraic Development Techniques 2001, Springer LNCS*. Genova, Italy (2001).
17. J. Padberg, H. Ehrig, L. Ribeiro. Algebraic High-Level Net Transformation Systems. In *Math. Struct. in Comp. Science 5*. Cambridge Univ. Press (1995), pp. 217–256.
18. J. Padberg, K. Hoffmann, M. Buder, A. Sünbül. Petri Net Modules for Component-Based Software Engineering. Technical Report, TU Berlin, 2001.
19. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, Singapore (1997).
20. M. Simeoni. *A Categorical Approach to Modularization of Graph Transformation Systems using Refinements*. PhD thesis, Dip. di Scienze dell'Informazione, Università di Roma *La Sapienza* (2000). Condensed version to appear in JCSS.
21. C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley (1997).
22. *Unified Modeling Language – version 1.3* (2000). Available at www.omg.org/uml.
23. M. Wermelinger, A. Lopes, J.L. Fiadero. A Graph Based Architectural Reconfiguration Language. In *Proc. ESEC/FSE '01*. ACM Press (2001).
24. A.M. Zaremski, J.M. Wing. Specification Matching of Software Components. In *ACM TOSEM '97*, pp. 333–369.