

# A Transformation-Based Component Framework for a Generic Integrated Modeling Technique

Hartmut Ehrig<sup>1</sup>, Fernando Orejas<sup>2</sup>,  
Benjamin Braatz<sup>1</sup>, Markus Klein<sup>1</sup>, and Martti Piirainen<sup>1</sup>

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

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

**Abstract.** This paper is based on two general ideas. The first one is the integration paradigm for data type and process modeling techniques developed by the first two authors within the last five years. The second one is a transformation-based component framework for system modeling presented at ETAPS 2002 in Grenoble. The aim of this paper is to join both ideas leading to a component framework for a generic integrated modeling technique. This component framework is based on transformations and is especially useful to be instantiated by graph- and net-based techniques. The main concepts are a self-contained semantics and internal correctness of components, based on a new idea of high-level constraints. Two main results concerning compositionality show that semantics and correctness for a system can be inferred from that of its components. The concepts are illustrated by a running example on modeling Java threads by high-level nets.

## 1 Introduction

In order to build up large software systems from smaller parts, a flexible component concept for software systems and infrastructures is highly important (see e.g. [29, 21, 14]). On the other hand, the integration of different kinds of data type and process modeling techniques is most important for software system specification in computer science and all kinds of applications. In our papers [9, 10] we have presented a conceptual and a formal model for an integration paradigm for this kind of modeling techniques. For a large variety of basic and integrated techniques (see [3, 17, 4, 20, 22, 24, 26, 28]) we have shown already how they can be interpreted and classified within this integration paradigm.

### 1.1 A Generic Component Framework for System Modeling

The aim of our paper [11] has been to present a generic component framework for system modeling that could support the development of the kind of systems

mentioned above. In this sense, we have defined a framework that can be used for a large class of semi-formal and formal modeling techniques. More precisely, we present 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. An import and an export connection relate these specifications. These connections are again generic, to allow a great variety of instantiations. We only require having suitable notions of inclusion and transformation (e.g. refinement) between specifications, 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. One of the key concepts 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 [12, 7, 26].

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. Moreover, this 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.

## 1.2 Main Concepts and Results of this Paper

In our component framework for system modeling in [11] we have not discussed the aspect that systems should be described from different views (data view, process view, ...). But this aspect has been widely discussed in several other papers leading to the conceptual and formal model for integrated data type and process modeling techniques in [10]. It is the main aim of this paper to join this concept with our component framework in [11].

This leads to a transformation-based component framework for a generic integrated modeling technique which is especially useful to be instantiated by graph- and net-based techniques. In this paper we review the generic component concept and also the integration paradigm, we pay special attention to the instantiation of our component framework in the context of this paradigm. Being specific, we introduce a notion of high-level constraints that allow us to present, in a unified way, specifications corresponding to any arbitrary modelling technique that can be integrated in our paradigm. Then, using this kind of specifications we are able to define components over these specifications, formulating a property of internal correctness, which is shown to be preserved under composition.

Finally, we briefly review how the various concepts can be instantiated to several modeling techniques. More specifically, we present a small case study using Petri nets to model Java threads [16], which is used as a running example also to illustrate an instantiation of our integration paradigm. For another, more detailed case study concerning a telephone service center we refer to [25], where the component concept for Petri nets in [25] can be considered as an instantiation of our generic framework.

### 1.3 Related Work

As mentioned above the generic integrated modeling technique is based on the integration paradigm in the papers [9, 10]. The generic component concept in [11] has been mainly motivated by the ideas in [21] for a component concept in the German BMBF project “Continuous Software Engineering” and by the module concepts for graph transformation systems in [27] and for Petri nets in [25]. In contrast to these concepts and that for UML [5] 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 [27, 25] is partly motivated by algebraic module specifications in [8]. The semantics of algebraic module specifications is based on free constructions between import and body part. This constructor-based semantics has been dropped in [25, 27] 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 [13, 32, 33, 1] could be considered complementary to ours. In particular, the use of graph transformation techniques proposed by Fiadeiro in [32] and also by Löwe in [19] 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 [21].

### 1.4 Organization of the Paper

The paper is organized as follows. In Sect. 2 we present some main requirements for a generic component framework which have guided our proposal in [9–11] and in this paper. Section 3 presents the main ideas of the basic framework in [11], introducing the component concept, its transformation semantics and a composition operation, showing the compositionality of semantics. Section 4 reviews the case-study of [11] that is used in the paper as a running example. Section 5 describes briefly our integration paradigm. Then, in Sect. 6, we study the instantiation of our component framework to integrated modeling techniques. Section 7 discusses the instantiation of the framework in the context of some specific modeling approaches. Finally, in Sect. 8, we present some conclusions,

discussing up to which point our framework meets the requirements presented in Sect. 2.

## 2 Main Requirements for a Generic Component Framework

In this section we present the requirements that, in our opinion, a component approach for integrated modeling techniques should satisfy.

### 2.1 Accordance with Current Component Technology

First of all, it should be clear that the proposed framework should be in accordance with the standard constructions used in current component technology for software development, enhancing the reusability of software and supporting system evolution ([29, 21]). This means that the proposed notion of component concept should make a clear distinction between the interface, that states how a given component is connected and used, and the body, where the services provided by the component are implemented. On the other hand, the framework should provide composition operations that allow us to put together components as it is done in standard software architectures.

### 2.2 Support of View-Oriented Semiformal and Formal Modeling Techniques

Secondly, the fact that the proposed framework should address the modeling phase poses some additional requirements. The key issue is that, in the modeling phase, we would typically want to be able to analyze and to reason about the single components and about the component-based systems. This means, in the first place, that we need to have a sufficiently precise semantics of the components and of the corresponding systems. However, we believe that different degrees of formality may be needed when modeling a certain system. In particular, when describing critical aspects of a system a formal technique is needed. But, when describing some less critical component, a semiformal approach could be sufficient. In this situation, the semantics of a component should be as precise as the component is. This means that a formal component should have a formal semantics. But, it should be possible to define some semiformal semantics for a component based on a semiformal modeling technique. We also believe that systems (and single components) are better modeled by, independently, describing them from different views (static, dynamic, behavior, ...). In this sense, the component framework should support this kind of view-oriented approach. Lastly, defining the semantics of a system compositionally, in terms of the semantics of the included components, would probably help in reasoning about the given system. In this sense, if a notion of internal correctness is defined for components, this property should be preserved by component composition.

### 2.3 Different Roles of Genericity

Finally, we think that genericity plays a fundamental role in a component framework. Genericity should be present in the framework in various senses. The most obvious one is that components must describe generic software units. The separation between the body and the interfaces in a component, discussed above, is the basis for this kind of genericity. Then, the component concept itself must be generic, in the sense that it should be possible to use it in connection with different modeling techniques, by means of an adequate “instantiation”. The reason is that component-based systems are often heterogeneous, i.e. include components based on different modeling formalisms (or programming languages). Hence, using a uniform notion of component, independently of the heterogeneity of the different contents, should simplify analyzing and reasoning about the system. Moreover, these tasks may be even more simplified if the semantics of a component is also generic, in the sense of being based on a number of basic concepts or constructs having a precise definition or instantiation on the modeling techniques considered.

## 3 Main Concepts for a Generic Component Framework

In this section, we present the main concepts for our generic component framework and show already, on a general level, how to satisfy the main requirements stated in the previous section. A simplified version without constraints has been given already in our paper [11]. In the following sections we give more detail how the basic notions of our framework can be interpreted in view of our integration paradigm. We start with some general assumptions concerning our modeling technique, which is one of the key generic concepts in our framework.

### 3.1 Generic Modeling Techniques with Constraints

We assume that a generic modeling technique is a general framework for describing systems. These descriptions have a syntactical part, consisting of signatures  $SIG$  and specifications  $SPEC$ , and a semantical part, consisting of behaviour or models  $M$  of the corresponding signature or specification, denoted  $M \in Mod(SIG)$  or  $M \in Mod(SPEC)$ , respectively. Moreover, it should be possible to deal with specific modeling approaches as concrete instances of the generic technique.

In order to express properties 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 in some logical formalism. We distinguish between loose and tight constraints. Loose constraints are defining a class of models, while tight constraints are defining a unique model up to isomorphism. Typical examples of loose constraints are predicate or temporal logic formulas. On the other hand, we show in Sect. 6 how a Petri net can be considered as a tight constraint in an integrated modeling technique.

A generic modeling technique in our framework allows us to formulate different types of tight and loose constraints. A signature  $SIG$  together with a single constraint or a class of constraints  $Constr$  is called a specification  $SPEC = (SIG, Constr)$ . We assume to have a notion of satisfaction for each kind of constraint, where  $M \models Constr$  means that  $M$  satisfies  $Constr$ . The class  $Mod(SPEC)$  consists of all models  $M \in Mod(SIG)$  such that  $M$  satisfies the constraints in  $Constr$ .

For verification purposes, a formal modeling technique should also have a suitable deduction calculus for constraints. From the software engineering point of view, we also require to have suitable horizontal and vertical structuring techniques, especially a notion of refinement or transformation of specifications.

### 3.2 A Generic Component Concept

Components are self-contained modeling units with a clear separation between the interface of the component and the body. Moreover, the interface can be divided into two parts: the import interface, describing what the component assumes about the environment, and the export interface, describing the services provided by the component itself.

In this sense, given a generic modeling technique with model specifications in the sense of 3.1, 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 3.3),  $exp: EXP \Rightarrow BOD$ , called export transformation, which is a *refinement* describing how the elements presented in the export interface are implemented by 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. We could have been slightly more general, by asking only for an inclusion of the import signature into the body signature, but not necessarily asking for the inclusion of the corresponding constraints. However, we feel that, dealing with this more general case, although technically not difficult, would slightly complicate the rest of the paper.

### 3.3 A Generic Transformation Concept

We need a generic transformation concept in order to formulate properties of export connections (see 3.2) and of connectors between import and export interfaces of different components (see 3.5, below). Again, we will try to be as general as possible.

A generic transformation  $trafo: SPEC_1 \Rightarrow SPEC_2$  between two model specifications  $SPEC_i = (SIG_i, Constr_i) (i = 1, 2)$  consists of transformations  $trafo_{SIG}: SIG_1 \Rightarrow SIG_2$  of the signatures and  $trafo_{Constr}: Constr_1 \Rightarrow Constr_2$  of the constraints.

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 extension property is called *compositional*, if the (chosen) corresponding extension diagrams are closed under horizontal and vertical composition. This is very similar – and in some instantiations equal – to the well-known composition property of pushouts in category theory.

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 [23]. 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 [12] 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.

### 3.4 Transformation Semantics of Components

According to the general requirements, components are 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.

The main idea proposed in [11] 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  $trafo': \Rightarrow SPEC'$ , 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  $TrafoSem(COMP)(trafo) = trafo' \circ exp \in Trafo(EXP)$ .

### 3.5 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, in a non-hierarchical system one may need an operation of circular composition, where the import of one component is the export of another component and vice versa. 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 and connectors in the sense of [1] may be more difficult to handle, however previous experience in dealing with module operations (see e.g. [8]) 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.

$$\begin{array}{ccccc}
 EXP_3 = EXP_1 \xrightarrow{exp_1} & BOD_1 & \xrightarrow{xconnect'} & BOD_3 & \\
 & \uparrow imp_1 & & \uparrow imp'_1 & \\
 IMP_1 \xrightarrow{connect} & EXP_2 & \xrightarrow{exp_2} & BOD_2 & \\
 & & & \uparrow imp_2 & \\
 & & & IMP_3 = IMP_2 & 
 \end{array}$$

**Fig. 3.** Composition of Components

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.



Using  $imp_3 = imp'_1 \circ imp_2$ ,  $exp_3 = xconnect' \circ exp_1$  and extension diagram (2)+(3) we have

$$TrafoSem(COMP_3)(trafo_3) = trafo_1 \circ xconnect' \circ exp_1 . \quad (5)$$

Concerning the right hand side of (4) we have

$$TrafoSem(COMP_2)(trafo_3) = trafo_2 \circ exp_2$$

and hence

$$\begin{aligned} & TrafoSem(COMP_1) \circ Trafo(connect) \circ TrafoSem(COMP_2)(trafo_3) \\ &= TrafoSem(COMP_1) \circ Trafo(connect)(trafo_2 \circ exp_2) \\ &= TrafoSem(COMP_1)(trafo_2 \circ exp_2 \circ connect) \\ &= trafo_1 \circ xconnect' \circ exp_1 , \end{aligned} \quad (6)$$

where the last step uses extension diagram (1)+(2).

Now (4) follows immediately from (5) and (6).  $\square$

In Sect. 6 we give a generic notion of internal correctness of components based on properties of constraints, such that the internal correctness is preserved by the composition operation.

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

As a small case study and running example we will model a few aspects of the behaviour of threads in the programming language Java with algebraic high-level Petri nets presented already in [11]. An overview reference for Java threads is given in [16].

We use a notation where an algebraic high-level net  $N$  consists of an algebraic signature  $\Sigma_N = (S, OP, X)$ , a  $\Sigma_N$ -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 and a transition  $tr$  itself is inscribed with a set of equations over  $\Sigma_N$ . In our example, however, we do not use equations for transitions.

We use a transformation concept similar to [24] 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.

### 4.1 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  $\Sigma_{EXP_1}$  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. 5. The type of all places is `Thread` and all arcs are inscribed with the variable  $t$  of sort `Thread`.

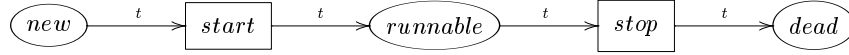


Fig. 5.  $EXP_1$

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. 6. 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`.

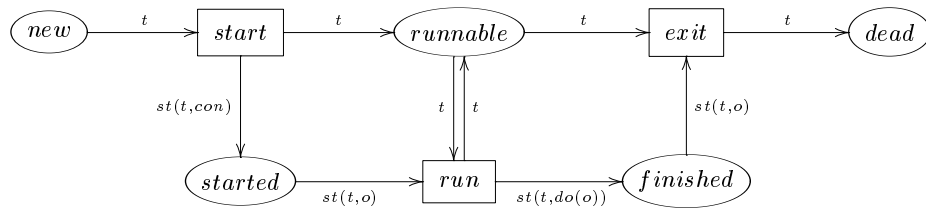


Fig. 6.  $BOD_1$

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

In the import interface (Fig. 7) only the *run*-transition and the adjacent places are kept, because this transition is useful to be further refined.

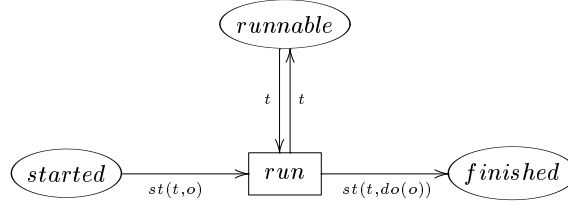


Fig. 7.  $IMP_1 = EXP_2$

#### 4.2 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  $do1, do2: Object \rightarrow Object$  to the signature. The *run*-transition is removed and replaced by two new transitions *act1* and *act2* with an intermediate place *working* (see Fig. 8). We assume that the algebra  $A_{BOD_2}$  satisfies the equation  $do2(do1(o)) = do(o)$  because sequential firing of *act1* and *act2* should still produce the same result as before.

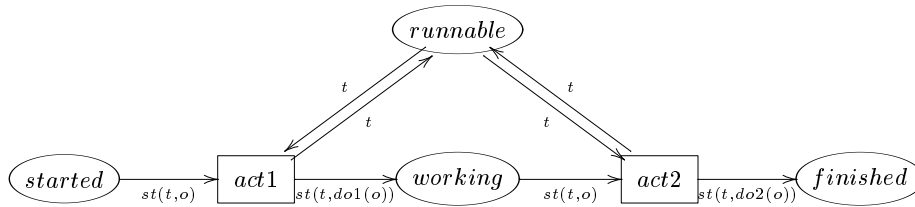


Fig. 8.  $BOD_2$

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:

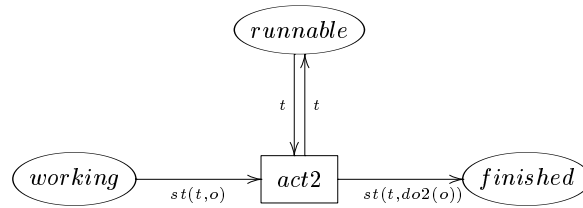
```
class MyThread2 extends MyThread {
    public void run() {
        act1();
    }
}
```

```

    act2();
  }
  private void act1() { ... }
  private void act2() { ... }
}

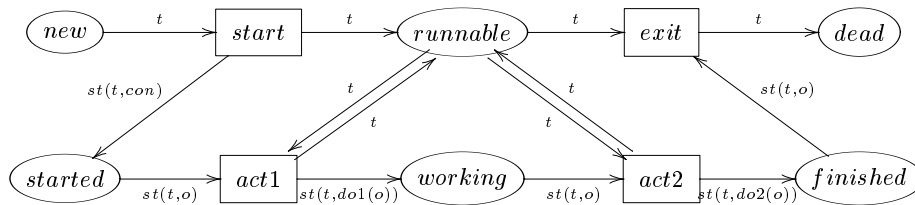
```

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. 9) with only the transition  $act2$  and the adjacent places.

Fig. 9.  $IMP_2$ 

### 4.3 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. 10) resulting from application of the rule underlying  $exp_2$  to the net  $BOD_1$ , replacing the transition  $run$  by  $act1$  and  $act2$ .

Fig. 10.  $BOD_3$ 

## 5 The Integration Paradigm

In this section we present some ideas about the integration paradigm, introduced in [9,10] to provide a uniform framework to deal with and integrate different data type and process modeling techniques. This paradigm is based on the use

of a specific kind of *integrated model* that can be used to give semantics to the different modeling techniques.

### 5.1 Conceptual Ideas of the Integration Paradigm

The aim of this semantic integration is to provide the basis for the integration of the tools and concepts associated to these techniques.

Integrated models can be seen as a semantic bottom-up description of given systems. This description is presented in four layers. The first one is the data type layer: we consider that the most basic elements of a system are the values processed, together with the basic operations defined over them. The second layer describes the data states and the intended state transformations. The intuition here is to model the internal states of the given system (or a system component) and the basic operations considered for their transformation. The third layer, the process layer, describes the intended reactive behaviour of a system. Finally, the fourth layer is the architecture layer, which describes the existing composition operations considered for building a system out of its components. Actually, the aim of this paper is to provide adequate concepts for the definition of this fourth layer. In this sense, in the rest of the section, we will mainly concentrate on the description of the first three layers, and how a given modeling technique is mapped into them.

A typical data type modeling approach, like algebraic specification or Z, will probably define integrated models including the first two layers (the process signature and the corresponding process semantics will probably be empty). Conversely, a typical (low level) process modeling approach, like CCS or place/transition nets, will have very simple components corresponding to the first and, perhaps, the second layer. Below we present in Fig. 11 very briefly how various modeling formalisms can be mapped into our integrated model. In [10] we have shown how some well-known formal modeling techniques, like CCS, place/transition nets, algebraic high-level nets, graph transformations and attributed graph transformations, can be seen as conceptual and formal instances of our paradigm.

### 5.2 Formal Concepts of the Basic Layers

Let us now introduce, in some more detail, the formal concepts for the basic three layers of our integration paradigm according to our paper [10].

1. The first layer, as said above includes the basic values and associated operations and relations used in a given system. This is represented by a signature  $\Sigma_0$ , on the syntactical side, and a (partial) data value  $\Sigma_0$ -algebra  $A_0$ , on the semantical side.
2. In our framework, data states are seen as algebras extending the basic data value algebra. This generalizes the cases where data states are tuples or some other kind of structured objects, like attributed graphs or labeled trees. This means that the second layer includes, on the one hand, a signature  $\Sigma$ ,

Specification Technique	Layer 1 Data Types	Layer 2 Data States & Transformations	Layer 3 Processes	Layer 4 System Architecture
<b>Algebraic High-Level Nets</b>	data tokens and algebras defined by algebraic specification	marking of places by data tokens & firing of transitions	net processes	parameterization, net transformations union / fusion
<b>Attributed Graph Transformation</b>	attributes and algebras defined by algebraic specification	attributed graphs and graph transformation	graph processes	parameterization, composition, modularization
$\mu$ <b>SZ</b>	type definition in $Z$	data states and operations schemas in $Z$	statechart processes	configurations
<b>UML</b>	basic data types defined by class diagrams	classes, attributes & methods	object-oriented statecharts & sequence diagrams	packages

Fig. 11. Integrated data type and process formalisms

extending  $\Sigma_0$ , providing a syntactic description of the attributes to access the values included in a given state, and a class of partial  $\Sigma$ -algebras  $DS$  defining the allowable data states of the given system. On the other hand, the second layer also includes a description of the state transformations available in the given system. We consider that transformations are mappings that have some (explicit) input parameters and are applied to data states (that act as implicit input parameters). The result provided by a transformation is a new state together with some output data values. This implies that this second layer includes a transformation signature:

$$T = \{T_{v;w}\}_{(v,w) \in S^* \times S^*}$$

where  $S$  is the set of sorts of the data state signature  $\Sigma$ . If  $t$  is in  $T_{s_1 \dots s_n; s'_1 \dots s'_m}$ , this means that  $t$  is the name of a transformation with input parameters of sorts  $s_1 \dots s_n$  and output parameters of sorts  $s'_1 \dots s'_m$ .

The semantics of a state transformation is represented by means of a *data state transition system*  $G_{DSTS}$  which includes all the possible transformations among all the possible states of the given system. In particular,  $G_{DSTS}$  is a graph whose nodes are labeled by data states and whose edges are labeled by transformation expressions, where a transformation expression is a term of the form  $t_{A,B}(a,b)$ , where  $t \in T_{s_1 \dots s_n; s'_1 \dots s'_m}$ ,  $a = (a_1, \dots, a_n) \in A_{s_1} \times \dots \times A_{s_n}$  and  $b = (b_1, \dots, b_m) \in B_{s'_1} \times \dots \times B_{s'_m}$ . In particular, if  $G_{DSTS}$  includes an edge from state  $A$  to state  $B$  with label  $t_{A,B}(a,b)$  this means that the transformation  $t$  with input parameters  $a$  can be applied to  $A$  yielding as a result the state  $B$  and output parameters  $b$ .

3. Similarly to transformations, we assume that processes may have input and output parameters. This means that, in our models, processes are represented

syntactically by a process signature:

$$P = \{P_{v;w}\}_{(v,w) \in S^* \times S^*}$$

On the other hand, semantically, the behavior of a process is represented by means of a *reactive state transition system*  $G_{RSTS}(p)$ , which is mapped by a partial graph morphism  $h(p)(a, b): G_{RSTS}(p) \rightarrow G_{DSTS}$  into the data state transition system  $G_{DSTS}$  where  $a$  and  $b$  are suitable input and output processes of process  $p$ .

### 5.3 Integrated Signatures and Models

Summarizing, integrated models consist of a syntactic part, the integrated signature  $M\text{-SIG}$ , and a semantic part  $M\text{-MOD}$ .  $M\text{-SIG}$  consists of the basic data type signature  $\Sigma_0$ , the state signature  $\Sigma$ , the transformation signature  $T$  and the process signature  $P$ . On the other hand,  $M\text{-MOD}$  consists of the basic data type algebra  $A_0$ , the class of data states  $DS$ , the data state transition system  $G_{DSTS}$ , and the processes semantics, consisting, for each process  $p$ , of the reactive state transition system  $G_{RSTS}(p)$ , the initial and final sets of nodes  $I(p)$  and  $F(p)$  and the families of morphisms  $h(p)$ . Given a model  $(M\text{-SIG}, M\text{-MOD})$ , with a certain abuse of terminology, we will often say that  $M\text{-MOD}$  is an integrated  $M\text{-SIG}$ -model, denoted  $M\text{-MOD} \in Mod(M\text{-SIG})$ .

As said above, our integration model can be used to give semantics to a large variety of data type or process specification or modeling formalisms, including the possibility of integrating some of them. The basic idea is that a given formalism can be seen as defining, for each specification, a certain integrated signature  $M\text{-SIG}$ , therefore defining a corresponding class of integration models,  $Mod(M\text{-SIG})$ . Then, the given specification (or, rather, the semantic description included) can be seen as restricting the corresponding class to the intended semantics of the specification.

### 5.4 Example of High-Level Nets in the Integration Paradigm

We take the algebraic high-level net  $BOD_1$  from our example in Sect. 4. It consists of an algebraic signature  $\Sigma_{BOD_1}$ , a  $BOD_1$ -algebra  $A_{BOD_1}$ , places  $Pl_{BOD_1}$ , transitions  $Tr_{BOD_1}$  and a *type*-function for places as well as a *pre*- and a *post*-function for transitions.

The data value signature of the integrated model of  $BOD_1$  is the signature  $\Sigma_0 = \Sigma_{BOD_1}$  of the net shown below. On the semantical side the data algebra is  $A_0 = A_{BOD_1}$ .

$\Sigma_0 =$  **sorts:** *Thread, RunState, Object*  
**opns:** *thread<sub>1</sub>, thread<sub>2</sub>: → Thread*  
*st: ThreadObject → RunState*  
*com: → Object*  
*do: Object → Object*

The data state signature

$$\Sigma = \Sigma_0 + \mathbf{opns}: \begin{array}{l} \text{new, runnable, dead: } \rightarrow \text{mult}(\text{Thread}) \\ \text{started, finished: } \rightarrow \text{mult}(\text{RunState}) \end{array}$$

extends  $\Sigma_0$  by one constant symbol for each place  $pl$ . The domain of this constant is the multiset of the sort  $\text{type}(pl)$ . The data states  $DS$  are all  $\Sigma$ -algebras extending  $A_0$ . These states are isomorphic to the markings of the net because the interpretation of the constant symbols in the  $DS$ -algebras models the tokens on the corresponding place.

The transformation signature

$$T = \mathbf{trafos}: \begin{array}{l} \text{start: Thread; } \lambda \\ \text{run, exit: Thread Object; } \lambda \end{array}$$

consists of one transformation symbol for each of the three transitions  $tr$ . The sorts of the input parameters are the types of the variables in  $\text{pre}(tr)$ , the sorts of the output parameters are the types of the remaining variables in  $\text{post}(tr)$ . In our example none of the transformation symbols has output parameters. In the data state transition system there is an edge  $A \xrightarrow{tr} B$  if the transition  $tr$  is activated in the marking corresponding to state  $A$  and the result of the firing is the marking that corresponds to  $B$ , thus  $G_{DSTS}$  is isomorphic to the marking graph of the net in Fig. 6.

For simplicity we do not examine processes of the net here but they are considered in [10]. Therefore the process signature is empty and there are no reactive state transition systems.

## 6 A Component Framework for a Generic Integrated Modeling Technique

In this section we come back to the main aim of this paper to combine the concepts for the generic component framework, presented in Sect. 3, with the integration paradigm, introduced in Sect. 5. We introduce the new concept of high-level constraints for integrated model signatures, which provides a framework for presenting, in a uniform way, several kinds of integrated model specifications. This is the basis for the instantiation of our generic component concept to this kind of integrated specifications, and allows us to define a notion of internal correctness of a component. Then, we prove that internal correctness is preserved under component composition.

In this section, for the sake of simplicity, we restrict ourselves to deal with “tight” modeling techniques, i.e. approaches where a given specification defines a unique model (up to isomorphism). This is the case of many modeling techniques, like Z, statecharts or Petri nets. As a consequence, we do not currently deal with “loose” approaches, like temporal logic specifications, where a given specification may have many (non-isomorphic) models. Nevertheless, it must be said that the constraints approach considered would also have allowed us to deal with these techniques in a uniform way, making use of the notion of loose constraints.

### 6.1 High-Level Constraints

Let  $\mathcal{HL}$  be a class of high-level structures, like a class of Petri nets or statecharts, together with an interpretation in the integration paradigm. This means that we have, for each high-level structure  $HL \in \mathcal{HL}$ , a signature  $Sig(HL)$  and a model  $Mod(HL)$ , such that  $Sig(HL)$  is an integrated model signature  $M-SIG$  and  $Mod(HL)$  is an integrated model  $M \in Mod(M-SIG)$ . Now, for each model signature  $M-SIG$ , we define the class of high level constraints for this signature by:

$$\mathcal{HL}\text{-Constr}(M-SIG) = \{HL \mid HL \in \mathcal{HL} \text{ and } Sig(HL) = M-SIG\}$$

For each integrated model  $M \in Mod(M-SIG)$  and each high-level constraint  $HL \in \mathcal{HL}\text{-Constr}(M-SIG)$ , we say that  $M$  satisfies  $HL$ , written  $M \models HL$ , if the model  $Mod(HL)$  is isomorphic to  $M$ , i.e.

$$M \models HL \iff Mod(HL) \cong M$$

We could be slightly more general by allowing  $M \in Mod(M-SIG')$  and defining:

$$M \models HL \iff Mod(HL) \cong RESTR(M)$$

where  $RESTR(M)$  is the restriction of  $M$  with respect to  $M-SIG$ . But, for the sake of simplicity, we only consider the simpler case in this paper. However, the more general case is important in view of the general requirement of system modeling with heterogeneous components based on different modeling techniques. In particular, let  $\mathcal{HL}_1$  and  $\mathcal{HL}_2$  be two different classes of high-level structures, like, for instance, Petri nets and Statecharts. Then, we could have high level constraints  $HL_1 \in \mathcal{HL}_1$  for  $M-SIG_1$ , and  $HL_2 \in \mathcal{HL}_2$  for  $M-SIG_2$ , where  $M-SIG_1$  and  $M-SIG_2$  are both included in  $M-SIG$ . Then, a model  $M \in Mod(M-SIG)$  would satisfy the high-level constraints  $HL_1$  and  $HL_2$  simultaneously, if the restrictions  $M_1$  of  $M$  to  $M-SIG_1$  and  $M_2$  of  $M$  to  $M-SIG_2$  would be isomorphic to the models  $Mod(HL_1)$  and  $Mod(HL_2)$ , respectively. This means that the subparts  $M_1$  and  $M_2$  of  $M$  are constrained by different modeling techniques.

### 6.2 Integrated Model Specifications with High-Level Constraints and Generic Transformations

An integrated model specification  $M-SPEC = (M-SIG, HL)$ , with a high-level constraint  $HL \in \mathcal{HL}\text{-Constr}(M-SIG)$  can be considered now as a special case of a model specification, in the sense of Sect. 3.1. Let us have a closer look at the generic transformation concept and the corresponding extension property (see 3.3) in this special case. A generic transformation  $trafo: M-SPEC_1 \Rightarrow M-SPEC_2$ , in this case, consists of the transformations  $trafo_{SIG}: M-SIG_1 \Rightarrow M-SIG_2$ , and  $trafo_{Cons}: HL_1 \Rightarrow HL_2$ , such that  $Sig(HL_1) = M-SIG_1$  and  $Sig(HL_2) = M-SIG_2$ . An inclusion  $i: M-SPEC_1 \subseteq M-SPEC_2$  consists of

$i_{SIG}: M-SIG_1 \subseteq M-SIG_2$ , an inclusion of signatures, and  $i_{Cons}: HL_1 \subseteq HL_2$ , an inclusion of  $\mathcal{HL}$ -structures.

The extension property means now that we have separate extension properties for integrated model signatures and  $\mathcal{HL}$ -structures, as shown in Fig. 12. They are, however, compatible in the sense that the corresponding pairs in each corner are integrated model specifications. This is especially valid for  $M-SPEC'_2 = (M-SIG'_2, HL'_2)$ , which means that the signature  $Sig(HL'_2)$  is equal to  $M-SIG'_2$ .

$$\begin{array}{ccc}
 M-SIG_1 & \xrightarrow{trafo_{SIG}} & M-SIG_2 \\
 \downarrow i_{1,SIG} & & \downarrow i_{2,SIG} \\
 M-SIG'_1 & \xrightarrow{trafo'_{SIG}} & M-SIG'_2
 \end{array}
 \qquad
 \begin{array}{ccc}
 HL_1 & \xrightarrow{trafo_{Cons}} & HL_2 \\
 \downarrow i_{1,Cons} & & \downarrow i_{2,Cons} \\
 HL'_1 & \xrightarrow{trafo'_{Cons}} & HL'_2
 \end{array}$$

**Fig. 12.** Extension diagrams for integrated model specifications

### 6.3 Internal Correctness of Components

The component framework of Sect. 3 can now be applied to integrated model specifications with high-level constraints. This means that we have all the constructions and results for components available in an even more sophisticated framework.

A *component*  $COMP = (IMP, EXP, BOD, imp, exp)$  consists now of integrated model specifications  $IMP$ ,  $EXP$  and  $BOD$  and import and export connections, defining an inclusion  $imp: IMP \subseteq BOD$  and a transformation  $exp: EXP \Rightarrow BOD$ .

This means, for the corresponding high-level constraints  $HL_{IMP}$ ,  $HL_{BOD}$  and  $HL_{EXP}$ , that we have the following properties, called *internal correctness* of the component:

1.  $HL_{IMP} \subseteq HL_{BOD}$
2.  $exp(HL_{EXP}) = HL_{BOD}$

where  $exp(HL_{EXP})$  is the result of the transformation  $exp_{Cons}: HL_{EXP} \Rightarrow HL_{BOD}$  of  $\mathcal{HL}$ -structures.

As remarked at the end of Sect. 3.2, we could have relaxed the assumption that the import connection is an inclusion of specifications, asking to have only an inclusion of signatures. This may be especially the case when the body is a tight specification, for instance a Petri net, and the import is a loose one, for instance consisting of a set of temporal logic formulas. In this situation we can also relax the internal correctness condition: We can just ask that the (transformed) export constraints should be implied by the import and the body constraints. Formally,  $(imp(Constr_{IMP}) + Constr_{BOD}) \rightarrow exp(Constr_{EXP})$ .

#### 6.4 Internal Correctness Preserved under Composition

In Sect. 3.5 we have defined the composition  $COMP_3$  of components  $COMP_1$  and  $COMP_2$ , using a connector  $connect: IMP_1 \rightarrow EXP_2$ . We have also assumed that the connector uniquely defines a transformation  $connect: IMP_1 \Rightarrow EXP_2$ . In the case of integrated model specifications with high-level constraints, this means especially that we have a transformation of  $\mathcal{HL}$ -constraints  $connect_{Cons}$  such that  $connect(HL_{IMP_1}) = HL_{EXP_2}$ .

We want to show that internal correctness of  $COMP_1$  and  $COMP_2$  implies that of the composition  $COMP_3$ .

**Theorem 2.** *Given internal correct components  $COMP_1$  and  $COMP_2$  and the composition  $COMP_3 = COMP_1 \circ_{connect} COMP_2$  then also  $COMP_3$  is internal correct.*

*Proof.* We have to show

$$HL_{IMP_3} \subseteq HL_{BOD_3} \text{ and } exp_3(HL_{EXP_3}) = HL_{BOD_3} . \quad (7)$$

The extension property in the case of  $\mathcal{HL}$ -constraints for the composition of components, in Fig. 3, means that, if we have

$$HL_{IMP_1} \subseteq HL_{BOD_1} \text{ and } xconnect(HL_{IMP_1}) = HL_{BOD_2} , \quad (8)$$

then we have also

$$HL_{BOD_2} \subseteq HL_{BOD_3} \text{ and } xconnect'(HL_{BOD_2}) = HL_{BOD_3} . \quad (9)$$

Now, by internal correctness of  $COMP_1$  and  $COMP_2$ , we have

$$HL_{IMP_1} \subseteq HL_{BOD_1} \text{ and } xconnect(HL_{IMP_1}) = exp_2 \circ connect(HL_{IMP_1}) = exp_2(HL_{EXP_2}) = HL_{BOD_2} ,$$

which implies (8). Hence, we have also (9) and can end the proof of (7) as follows:

$$HL_{IMP_3} = HL_{IMP_2} \subseteq HL_{BOD_2} \subseteq HL_{BOD_3} \text{ and } exp_3(HL_{EXP_3}) = xconnect'(exp_1(HL_{EXP_1})) = xconnect'(HL_{BOD_1}) = HL_{BOD_3} .$$

□

#### 6.5 Example of Components in the Framework of Integrated Model Specifications

As an example we will transform the component  $COMP_1$  from our running example in Sect. 4 into the framework of integrated model specifications.

The export interface now is a model specification  $EXP$  containing a model signature  $M-SIG_{EXP}$  and the algebraic high-level net  $HL_{EXP} = EXP_1$  from Sect. 4.1 as constraint. The signature  $M-SIG_{EXP}$  is constructed as shown in Sect. 5.4. The specifications  $BOD$  and  $IMP$  are achieved correspondingly.

The export transformation  $exp: EXP \Rightarrow BOD$  now consists of two transformations  $exp_{SIG}: M-SIG_{EXP} \Rightarrow M-SIG_{BOD}$  and  $exp_{Cons}: HL_{EXP} \Rightarrow HL_{BOD}$ . The latter is the same transformation as  $exp_1$  from Sect. 4.1, while the former just does the corresponding transformations on the syntactical side, i. e. removes the transformation  $stop$  from  $T_{EXP}$  and adds the sorts *RunState* and *Object* and the operations  $st$ ,  $con$  and  $do$  to  $\Sigma_0$ , the constants *started* and *finished* to  $\Sigma$  and the transformations  $run$  and  $exit$  to  $T$ .

The import inclusion  $imp$  consists of  $imp_{Cons}: HL_{IMP} \subseteq HL_{BOD}$ , which is the same inclusion as  $imp_1$  from Sect. 4.1, and the corresponding inclusion on the syntactical side  $imp_{SIG}: M-SIG_{IMP} \subseteq M-SIG_{BOD}$ .

The resulting component is internally correct since  $HL_{IMP} \subseteq HL_{BOD}$  by the inclusion  $imp_{HL}$  and  $exp(HL_{EXP}) = HL_{BOD}$ .

## 7 Instantiations of the Generic Component Framework

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, including especially the case of integrated modeling techniques. This is reflected by our generic notion of integrated model signatures and models. 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. Finally, the framework is generic with respect to the kind of constraints used for modeling techniques, allowing us to formulate internal correctness of components.

In this section, we will briefly review some examples of instantiations for all these generic aspects, i.e., for integrated model signatures, transformations and constraints. Moreover, we discuss instantiations of the generic component framework leading to existing and new component concepts for specific integrated modeling techniques. This includes component concepts for Petri nets, graph transformations and some ideas for other visual modeling techniques, like UML.

### 7.1 Instantiations of Integrated Modeling Signatures

In Sect. 5, we have reviewed our integration paradigm, including conceptual and formal instantiation for several data type and process modeling techniques. Especially, formal instantiations have been given in our paper [10] for CCS, place/transition nets, algebraic high-level nets, graph transformations and attributed graph transformations. Moreover, conceptual instantiations have been discussed for  $\mu SZ$ , statecharts, UML class diagrams and others in [9].

### 7.2 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 [12, 7]. The extension property for transformations formulated in Sect. 3 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 algebraic specifications and algebraic high-level 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. At least, the horizontal and vertical composition property, required in 3.6, still has to be formulated 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 Sect. 4.

### 7.3 Instantiations of High-Level Constraints

As discussed in Sect. 6.1, high-level constraints can be formulated for any kind of integrated modeling technique, which can be instantiated in the integration paradigm. This has been done for all the formal integrated modeling techniques discussed in 7.1. On the other hand, in order to be able to instantiate our generic component framework for these modeling techniques transformations of high-level constraints satisfying the extension property need to be defined (see 6.2). This means that we can apply our framework to all the formal integrated modeling techniques, which have not only an interpretation in the integration paradigm, but which are also instances of high-level replacement systems as discussed in subsection 7.2. Especially, this has been done for different kinds of Petri nets and algebraic specifications.

### 7.4 Component Concepts for Petri Nets

As discussed above, Petri nets have been interpreted in the formal model of our integration paradigm. Moreover, 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. 3. 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. 4, including composition of components. This component concept for Petri nets is closely related to the component concepts for Petri nets presented in [25]. In fact, refinements of transitions, in the sense of [25], are closely related to rule-based transformations of Petri nets, where the left-hand side of the rule consists of a single transition only. Especially, it seems to be possible to consider the case study in [25], in the application domain of telephone services, also as a case study of our component concept.

### 7.5 Component Concepts for Graph Transformation Systems

Similar to the case of Petri nets discussed above, also graph transformation systems have been interpreted in the formal model of our integration paradigm. M. Simeoni ([27]) 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 [27], can be considered as transformations in the sense of 3.3, because they are shown to satisfy the extension property. More precisely, it is shown in [27] 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 ([15]) and local action systems in the sense of [18]. In both cases this means that the extension diagram is a special kind of pushout considered in [27]. As pointed out in the introduction the syntactical concept of the module concept for graph transformation systems in [27] is the same as that of our component concept in Sect. 3, instantiated for graph transformation systems. In [10] we discussed already an extension towards attributed graph transformations ([20]) which leads now to a new component concept for this case.

### 7.6 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 ([20]). In [10], we have discussed already how attributed graph transformations can be interpreted in our integration paradigm. Moreover, in [2], 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 ([30]), where at least 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 [6], 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 7.2. This seems to be a good basis for a component concept for statecharts.

## 8 Conclusion

We have presented a transformation-based component framework for system modeling. More precisely the main aim of this paper is a component framework for a generic integrated modeling technique combining the integration paradigm in [9, 10] with the generic component framework in [11]. This framework is based on a generic transformation concept, which is used to express intra- and inter-dependencies of interfaces and bodies of components given by integrated model specifications. We have introduced a new type of constraints for integrated specifications, called high-level constraints, which allows us to use high-level Petri nets or other kinds of high-level structures as constraints.

Finally let us discuss in which way our generic component framework meets the main requirements stated in Sect. 2. Our proposed framework meets to a large extent the first requirement to be in accordance with standard constructions used in current component technology as discussed in 2.1. However, partly due to lack of space, in this paper we just study a simple operation of (hierarchical) composition of components. However, we have also 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 [13, 32, 19] discussed in the introduction.

Concerning the requirements in 2.2 we have defined components not only as syntactical, but also as self contained semantical units. We have shown as two basic results that our semantics based on transformations is compositional and that internal correctness of the system model can be inferred from its components.

The genericity requirements in 2.3 have been the main guidelines for our generic framework. Especially we have shown in Sect. 7 how to instantiate all the generic parameters. Further exploration of the new idea of high-level constraints should be especially useful for system modeling with heterogeneous components, based on different modeling techniques. However, presently, it remains open how to combine in our framework different kinds of high-level constraints and well-known examples of loose constraints, like formulas in predicate or temporal logic.

Finally let us point out again that our generic framework is suitable not only for formal modeling techniques mainly discussed in this paper, but also for semiformal 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 ([5]).

**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).

## 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. R. Büsow, R. Geisler, and M. Klar. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. In *FASE '98*, pp. 71–87. Springer LNCS 1382 (1998).
4. E. Brinksma, editor. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard ISO 8807 (1989).
5. J. Cheesman, J. Daniels. *UML Components*. Addison-Wesley (2001).
6. H. Ehrig, R. Geisler, M. Klar, J. Padberg. Horizontal and Vertical Structuring for Statecharts. In *Proc. CONCUR '97*, pp. 327–343. Springer LNCS 1301 (1997).
7. 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*, pp. 361–404. Cambridge Univ. Press (1991).
8. 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).
9. H. Ehrig and F. Orejas. Integration Paradigm for Data Type and Process Specification Techniques. In *Bull. EATCS 65, Formal Spec. Column, Part 5* (1998).
10. 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*, pp. 201–228. Heraclion, Greece (2001). Also in *Electronic Notes in Theor. Comp. Science 50,3* (2001).
11. H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A Generic Component Concept for System Modeling. In *FASE 2002*.
12. H. Ehrig, M. Pfender, H. Schneider. Graph Grammars: An Algebraic Approach. In *Proc. SWAT '73*, pp.167 - 180.
13. J.L. Fiadero, A. Lopes. Semantics of Architectural Connectors. In *Proc. TAPSOFT '97*, pp. 505–519. Springer LNCS 1214 (1997).
14. V. Gruhn, and A. Thiel. *Komponentenmodelle: DCOM, JavaBeans, Enterprise-JavaBeans, CORBA*. Addison-Wesley (2000).
15. 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.
16. C.S. Horstmann, and G. Cornell. *Core Java 2. Volume II – Advanced Features*. Sun Microsystems Press, Prentice Hall PTR (2000).
17. D. Harel, and E. Gery. Executable Object Modeling with Statecharts. In *IEEE Computer, vol. 30, no. 7*. IEEE (1997).
18. D. Janssens, N. Verlinden. A Framework for ESM and NLC: Local Action Systems. In *Springer LNCS 1764* (2000), pp. 194–214.
19. M. Löwe. Evolution Patterns. In *Proc. Conf. on Systemics, Cybernetics and Informatics 1999, vol. II*, pp. 110–117.
20. 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.
21. 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).

22. J. Meseguer, and U. Montanari. Petri Nets are Monoids. In *Information and Computation*, 88(2), pp. 105–155 (1990).
23. 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).
24. 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.
25. J. Padberg, K. Hoffmann, M. Buder, A. Sünbül. *Petri Net Modules for Component-Based Software Engineering*. Technical Report, TU Berlin (2001).
26. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, Singapore (1997).
27. 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.
28. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall (1992).
29. C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley (1997).
30. *Unified Modeling Language - version 1.3* (2000). Available at [www.omg.org/uml](http://www.omg.org/uml).
31. H. Weber, H. Ehrig. Specification of Modular Systems. In *IEEE Trans. Software Eng. vol. SE-12* (1986), pp. 784–798.
32. M. Wermelinger, A. Lopes, J.L. Fiadero. A Graph Based Architectural Reconfiguration Language. In *Proc. ESEC/FSE '01*. ACM Press (2001).
33. A.M. Zaremski, J.M. Wing. Specification Matching of Software Components. In *ACM TOSEM '97*, pp. 333–369.