

A Component Framework Based on High-Level Replacement Systems

Hartmut Ehrig^{a,1}, Fernando Orejas^{b,2}, Benjamin Braatz^a,
Markus Klein^a and Martti Piirainen^a

^a *Technische Universität Berlin,
Franklinstraße 28/29, 10587 Berlin, Germany*

^b *Universitat Politècnica de Catalunya,
Campus Nord, Mòdul C6, Jordi Girona 1-3, 08034 Barcelona, Spain*

Abstract

This paper is based on two general concepts. The first one is a generic component framework for system modeling presented at FASE 2002, which is especially useful for graph- and net-based modeling techniques. The second one is the concept of high-level replacement systems, which has been studied within the last decade as an abstraction of the DPO-approach for graph transformation systems in a categorical framework, with instantiations to a large class of different modeling techniques. In this contribution both concepts are combined in the sense that the generic transformation concept – essentially used in the component framework – is instantiated by high-level replacement systems. As the main result we show how the properties for transformations required in the component framework can be shown in the case of high-level replacement systems. Moreover, some interesting extensions concerning multiple interfaces, union, and operational semantics of components are proposed.

1 Introduction

In [3] we presented a very generic notion of component, whose semantics is based on an equally generic notion of transformation, which is especially useful for graph transformations and visual modeling techniques. In particular, in [3] we discuss the application of this approach to a number of visual modeling techniques. The aim of such a generic approach was to provide unifying concepts that could be used to model (and to reason about) heterogeneous systems such as the ones supported by heterogeneous platforms like CORBA or COM+.

¹ Email: ehrig@cs.tu-berlin.de

² Email: orejas@lsi.upc.es

In this paper, we instantiate the framework presented in [3] by considering that transformations are defined by the application of high-level replacement rules ([1]), these are double pushout rules generalized from graphs to objects in suitable categories, which can be instantiated to different kinds of high-level structures. It can be noted that the approach still remains very generic, since many kinds of transformations can be seen as special cases of high-level rule transformations. The aim of this instantiation is to be able to study some constructions that need a more concrete framework than the one provided in [3]. In this sense, in this paper we propose some extensions to the basic constructions and results presented in [3]. In particular, we study a variation on the component concept by considering components including several import and export specifications. Moreover, we present an operation of union of components that was difficult to define at the level of [3]. Finally, we propose how to relate the transformation semantics of a component and the operational semantics (defined in terms of computations) of the specifications involved.

The paper is organized as follows. In Section 2 we briefly recall the main concepts introduced in [3]. Section 3 is dedicated to define the instantiation of these concepts to the case of high-level replacement transformations. Section 4 presents an example of a component using place/transition nets. In Section 5 components with multiple interfaces are introduced and the example is enhanced by a partial composition with a second component. In Section 6 we propose the two further extensions to our framework mentioned above. Finally, Section 7 provides some concluding remarks.

2 The Generic Component Framework

Components are self-contained units, where some details are hidden to the external user. This is achieved by providing a clear separation between the interface of the component and the body. The interface consists of 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. Obviously, the import and export interfaces are connected to the body in some well-defined way.

A *component specification*, in short *component*, is a 5-tuple:

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

where *IMP*, *EXP*, and *BOD* are three specifications called, respectively, the *import interface*, the *export interface*, and the *body*. Then, $imp: IMP \rightarrow BOD$, and $exp: EXP \rightarrow BOD$, are two connections called, respectively, the *import connection*, and the *export connection*.

This notion leaves open the modelling technique used to describe the specifications involved and the kind of connectors used to relate the interfaces and

body. Intuitively, we assume that the import connection is some kind of inclusion, in the sense that the functionality defined in the body is built upon the import interface. We also assume that the export connection is some kind of transformation describing a *refinement* of the export interface.

Semantically, a component builds a transformation (refinement) of the export interface from each given transformation of the import interface. More precisely, we 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.

To formulate this definition properly, we must impose certain requirements on the kinds of transformations considered. 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 $i_1: SPEC_1 \hookrightarrow SPEC'_1$ there is a selected transformation $trafo': SPEC'_1 \Rightarrow SPEC'_2$ with inclusion $i_2: SPEC_2 \hookrightarrow SPEC'_2$, called the *extension* of $trafo$ with respect to i_1 , leading to the extension diagram in Figure 1.

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

Fig. 1. Extension diagram for the extension property

It must be pointed out that, in a given framework \mathcal{T} , given $trafo$ and i_1 as above, there may be several $trafo'$ and i_2 , that could satisfy this extension property. Our assumption means that only one such $trafo'$ and i_2 are chosen, in some well-defined way, as the extension of $trafo$ with respect to i_1 . We could have also required that these extensions only exist when the given $trafo$ is consistent with i_1 in a specific sense. This is the case in the section below.

Essentially, this extension property means that if one can apply a transformation on a certain specification, then it should be possible to apply the “same” transformation on a larger specification.

Now we can define the semantics of a component following the ideas described above. Let us denote by $Trafo(SPEC)$ the class of all transformations $trafo: SPEC \Rightarrow SPEC'$ from $SPEC$ to some specification $SPEC'$. The *transformation semantics* of the component $COMP$ is defined as a function

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

where, according to Figure 2, for all $trafo \in Trafo(IMP)$:

$$TrafoSem(COMP)(trafo) = trafo' \circ exp \in Trafo(EXP)$$

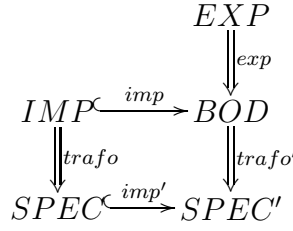


Fig. 2. Transformation semantics

Several operations on components can be considered in our generic framework. In [3] we have only defined a basic composition operation for connecting the import of one component with the export of another component. Again, we see this connection as a transformation:

Given components $COMP_1$ and $COMP_2$ and a transformation, $connect: IMP_1 \Rightarrow EXP_2$, we 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 $imp'_1: BOD_2 \hookrightarrow BOD_3$ in Figure 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 $exp_3 = xconnect' \circ exp_1$.

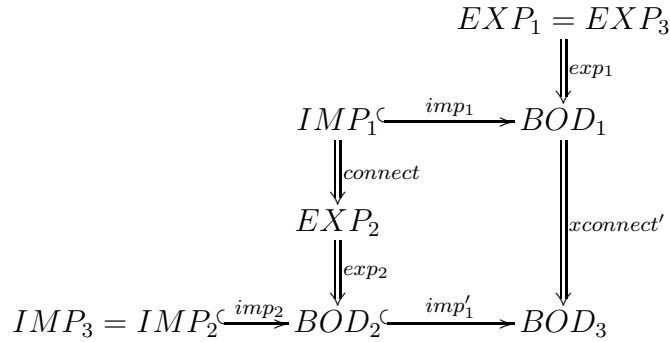


Fig. 3. Composition of components

The semantics of this composition operation can be shown to be compositional if the horizontal and vertical composition of extension diagrams, as given in Figure 1, is again an extension diagram. In particular, in this case

$$TrafoSem(COMP_3) =$$

$$TrafoSem(COMP_1) \circ Trafo(connect) \circ TrafoSem(COMP_2)$$

where $Trafo(connect)(trafo) = trafo \circ connect$.

3 Instantiation to High-Level Replacement Systems

In this section we present high-level replacement systems, short HLR systems as an important instantiation of our component framework. HLR systems, as an abstraction of graph transformation systems, were introduced in [1]. This abstraction is obtained by defining HLR systems for any category \mathbf{CAT} with a start object $S \in |\mathbf{CAT}|$ and a set of rules.

Several results concerning concurrency and parallelism have been proven in [1]. Most of these theorems need certain properties, called HLR conditions, to become valid. As a consequence, instances of HLR systems are often examined concerning these HLR conditions. In [1] it is also shown that in addition to various kinds of graph transformation systems also algebraic specifications and Petri nets are valid instantiations of HLR systems.

According to the first formulation of HLR systems we will use the double pushout approach to express rules and rule applications. This means that a rule consists of three objects and two morphisms $L \leftarrow K \rightarrow R$. A direct transformation of an object G_0 according to a rule is given by a context object C and a morphism $K \rightarrow C$, such that G_0 becomes a pushout object for diagram (1) in Figure 4. The result of the transformation is then given by G_1 as a pushout object for diagram (2).

$$\begin{array}{ccccc}
 L & \longleftarrow & K & \longrightarrow & R \\
 \downarrow & & \downarrow & & \downarrow \\
 & (1) & & (2) & \\
 G_0 & \longleftarrow & C & \longrightarrow & G_1
 \end{array}$$

Fig. 4. Double pushout

Transformations in general are then defined by sequences of direct transformations. For the instantiation of the component framework it is sufficient to show that HLR systems satisfy the extension property mentioned in Section 2.

In the case of HLR systems the transformation and the corresponding embedding have to be consistent in a certain sense. In order to express this condition in categorical terms we need the notion of *initial pushouts*, which are explicitly defined in [7].

The needed consistency can be formulated as:

Definition 3.1 A morphism $k: G_0 \rightarrow G'_0$ is *consistent* with respect to a transformation $G_0 \xrightarrow{R} G_n$, if:

- (i) There is an initial pushout with respect to $k: G_0 \rightarrow G'_0$, given by diagram (1) in Figure 5.
- (ii) There are morphisms $j_i: Bound \rightarrow C_i$, for $1 \leq i \leq n$, such that:
 - (a) $cl_1 \circ j_1 = b$
 - (b) $cl_i \circ j_i = cr_{i-1} \circ j_{i-1}$, for $1 < i \leq n$,

where n denotes the number of direct transformations in p and C_i the image

of the interface of the i -th direct transformation in p . $cl_i: C_i \rightarrow G_{i-1}$ and $cr_i: C_i \rightarrow G_i$ denote the corresponding morphisms of the i -th direct transformation in p .

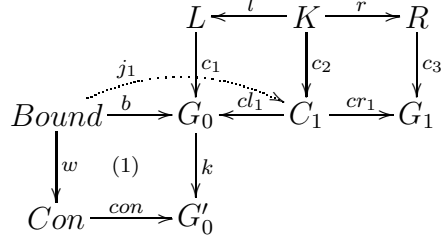


Fig. 5. Consistent embedding

The diagram in Figure 5 shows in the case $n = 1$ the needed objects and morphisms for a consistent morphism k with respect to a transformation, that consists of one direct transformation, where (1) is supposed to be an initial pushout.

Theorem 3.2 *Given a transformation $G_0 \xrightarrow{p} G_n$ and a morphism $k: G_0 \rightarrow G'_0$, such that k is consistent with respect to p , then we obtain a selected transformation $G'_0 \xrightarrow{p'} G'_n$ and a selected morphism $k': G_n \rightarrow G'_n$.*

This embedding theorem is well-known in the graph case. For the HLR case a similar result can be found in [7]. In order to obtain the compositionality result discussed in Section 2 we need to show horizontal and vertical composition of extension diagrams.

Theorem 3.3 *Given the two extension diagrams in Figure 6, where d_i is consistent with respect to $G_{i-1} \xrightarrow{p_i} G_i$ ($i = 1, 2$), then the composition is also an extension diagram where d_1 is consistent with respect to the transformation $G_0 \xrightarrow{p_1; p_2} G_2$.*

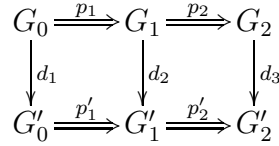


Fig. 6. Horizontal composition of extension

A similar result holds for the vertical composition of extension diagrams. The results in this section can be instantiated to all kinds of HLR systems with suitable pushouts. We are especially interested in different kinds of Petri nets, which are outlined in the next section.

4 Example of a Petri Net Component

As a small example we now present a model of the well-known producer/consumer system using place/transition nets and net transformations as defined for high-level nets in [6]. An example using high-level nets can be found in [3]. The component *PC-COMP* inserts a buffer between the producer and the consumer of the system.

The export interface *PC-EXP* of *PC-COMP* in Figure 7 specifies a producer/consumer system without a buffer, i.e. producing and consuming must occur synchronously (modeled by the transition *prodcons*).

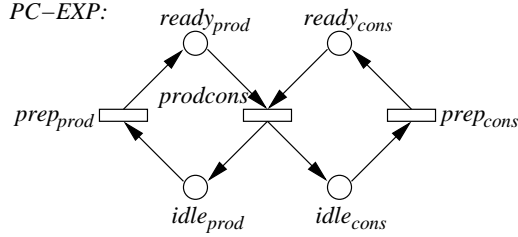


Fig. 7. Export *PC-EXP* of *PC-COMP*

In Figure 8 we define a HLR rule p_{buffer} that inserts a buffer.

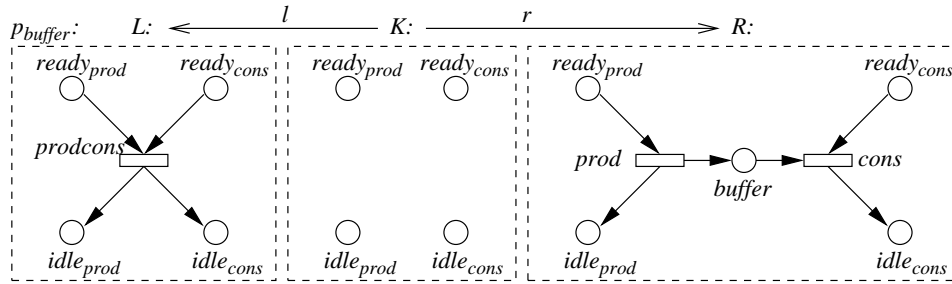
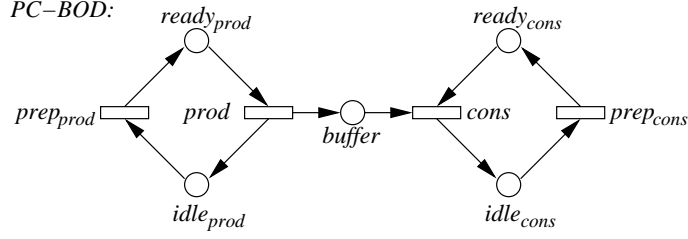
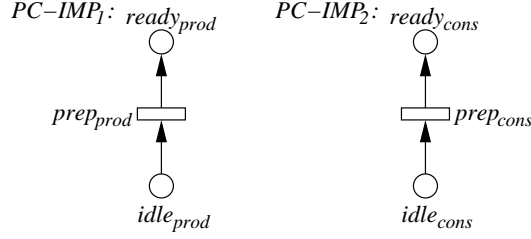


Fig. 8. Rule for export connection of *PC-COMP*

Applying p_{buffer} to the net *PC-EXP*, using the match morphism $m: L \rightarrow PC-EXP$ that is given by identities on the names, sets up a double-pushout (DPO) diagram with a resulting net *PC-BOD*. This DPO defines a transformation $exp: PC-EXP \Rightarrow PC-BOD$, establishing the body of *PC-COMP* in Figure 9. In this net, producing and consuming are decoupled; several tokens can be produced in a row before one is consumed. Introducing such a buffer is a typical way to improve the performance of a concurrent system.

As an example of multiple interfaces introduced in the next section, we define in Figure 10 two import interfaces, each a subnet of *PC-BOD*, in order to leave parts of the specification open. The transitions $prep_{prod}$ and $prep_{cons}$ should be refined by more complex nets when composing *PC-COMP* with other components.

Thus, we have $PC-COMP = (PC-IMP, PC-EXP, PC-BOD, imp_{PC}, exp_{PC})$, where the import interfaces are $PC-IMP = (PC-IMP_1, PC-IMP_2)$


 Fig. 9. Body $PC-BOD$ of $PC-COMP$

 Fig. 10. Import $(PC-IMP_1, PC-IMP_2)$ of $PC-COMP$

and the import connections $imp_{PC} = (imp_{PC,1}, imp_{PC,2})$ are the inclusions of the import nets into $PC-BOD$.

5 Multiple Interfaces

In this section we extend the generic component framework of Section 2 by multiple interfaces. We show how this can be realized in the HLR-framework of Section 3, and how the example of Section 4 can be enhanced by a partial composition with a second component.

5.1 General Concept

For several applications it is useful to have explicitly several import and export interfaces for components in contrast to Section 2. This can be modeled by a multi-interface component $MI-COMP = (IMP, EXP, BOD, imp, exp)$, where we have now $IMP = (IMP_i)_{i=1,\dots,n}$, $EXP = (EXP_j)_{j=1,\dots,m}$, $imp = (imp_i: IMP_i \hookrightarrow BOD)_{i=1,\dots,n}$ and $exp = (exp_j: EXP_j \Rightarrow BOD)_{j=1,\dots,m}$.

The transformation semantics in this case is a function

$$TrafoSem(MI-COMP): \prod_{i=1}^n Trafo(IMP_i) \longrightarrow \prod_{j=1}^m Trafo(EXP_j)$$

where each family $(trafo_i: IMP_i \Rightarrow SPEC_i)_{i=1,\dots,n}$ is mapped to the family $(trafo' \circ exp_j: EXP_j \Rightarrow SPEC')_{j=1,\dots,m}$ with $trafo'$ defined by the diagram in Figure 11.

For the construction of this diagram we have to assume not only the extension property, but also the existence of a unique inclusion imp and a unique

$$\begin{array}{ccc}
 \sum_{i=1}^n IMP_i & \xrightarrow{imp} & BOD \\
 \Downarrow trafo & & \Downarrow trafo' \\
 \sum_{i=1}^n SPEC_i & \xrightarrow{imp'} & SPEC'
 \end{array}$$

Fig. 11. Extension diagram for multiple interfaces

transformation $trafo$ induced by the families $(imp_i)_{i=1,\dots,n}$ and $(trafo_i)_{i=1,\dots,n}$ respectively. The existence of the inclusion imp is equivalent to the disjointness of the images of the import interfaces in BOD , while the existence of the transformation $trafo$ has to be provided by the transformation framework.

In our instantiation to HLR systems, we can construct this transformation for given transformations $trafo_i$ via rules r_i and match morphisms h_i for $i = 1, \dots, n$ by parallel application of the rules r_i via match morphisms $in_i \circ h_i$, where $in_i: IMP_i \hookrightarrow \sum_{i=1}^n IMP_i$ is the inclusion of IMP_i into the coproduct.

Concerning the composition of multi-interface components $MI-COMP_1$ and $MI-COMP_2$ we are now able to have a connection $connect: IMP_{1,i_0} \Rightarrow EXP_{2,j_0}$ for only two specific members of the corresponding import and export interface families. The body BOD_3 of the composition $MI-COMP_3$ is defined similar to Figure 3. As export interfaces of $MI-COMP_3$ we only take those of $MI-COMP_1$. The family of import interfaces of $MI-COMP_3$ consists of all $IMP_{1,i}$ for $i \neq i_0$ and all $IMP_{2,j}$, where we assume that $imp_{1,i}: IMP_{1,i} \hookrightarrow BOD_1$ for $i \neq i_0$ can be extended to $imp_{3,i}: IMP_{1,i} \hookrightarrow BOD_3$.

In the case of HLR-transformations this extension is possible, because the images of the unused import interfaces $IMP_{1,i}$ with $i \neq i_0$ are disjoint from the image of IMP_{1,i_0} and are therefore preserved by the application of the transformation.

Under suitable conditions it is also possible to achieve a compositionality result for semantics in the multiple case similar to that in Section 2.

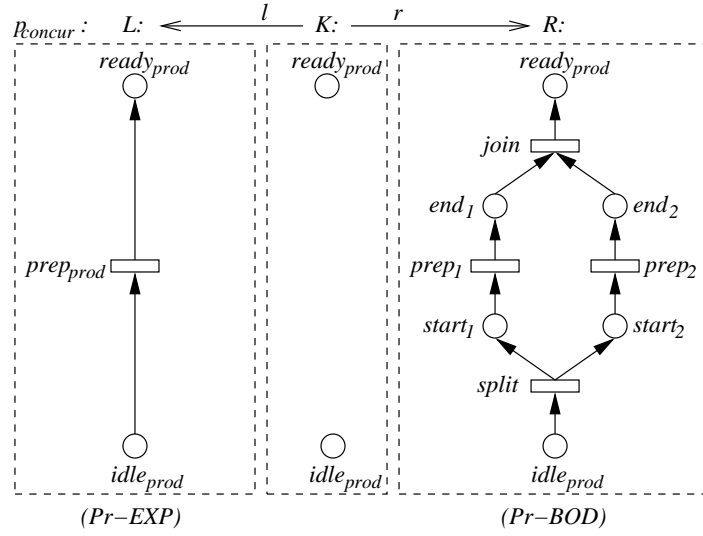
5.2 Example of Partial Composition

In order to show partial composition we extend our example by a second component $Pr-COMP$ which further refines the preparation phase of the producer.

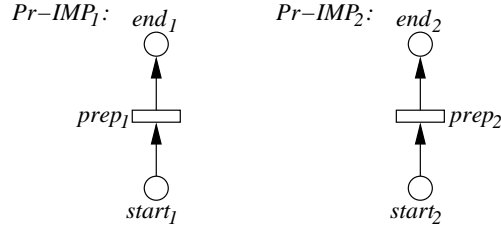
The export interface $Pr-EXP$ of this component $Pr-COMP$ is identical to the left hand side of the rule p_{concur} shown in Figure 12, which replaces the transition $prep_{prod}$ representing the preparation phase by a subnet with two concurrent transitions $prep_1$ and $prep_2$.

Hence, the match morphism for the application of the rule p_{concur} to $Pr-EXP$ is the identity, the export transformation exp_{Pr} is given by this application and the body $Pr-BOD$ is isomorphic to the right hand side of p_{concur} .

Again, we define two import interfaces $Pr-IMP_1$ and $Pr-IMP_2$ in order to allow the transitions $prep_1$ and $prep_2$ to be refined independently (Figure


 Fig. 12. Rule for export connection of $Pr-COMP$

13).


 Fig. 13. Import of $Pr-COMP$

Considering Figure 14 we can now construct the partial composition $Comp-COMP$ via the identical connector $connect: PC-IMP_1 \Rightarrow Pr-EXP$.

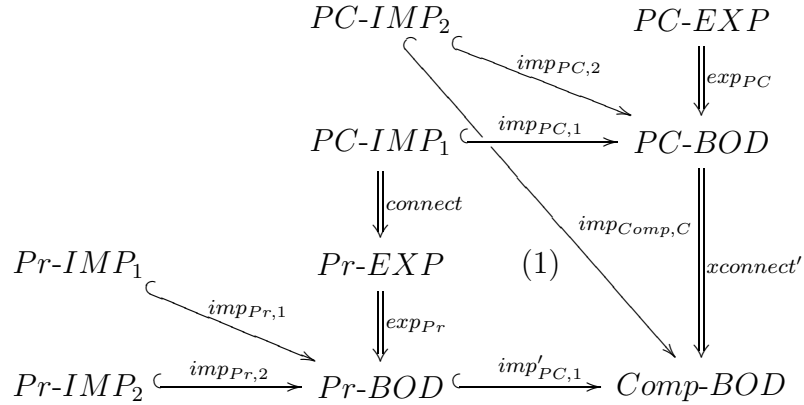


Fig. 14. Partial composition

The export interface of $Comp-COMP$ is given by $PC-EXP$ with the export transformation $exp_{Comp} = xconnect' \circ exp_{PC}$, where $xconnect'$ and the

body *Comp-BOD* (Figure 15) are given by extension diagram (1) in Figure 14.

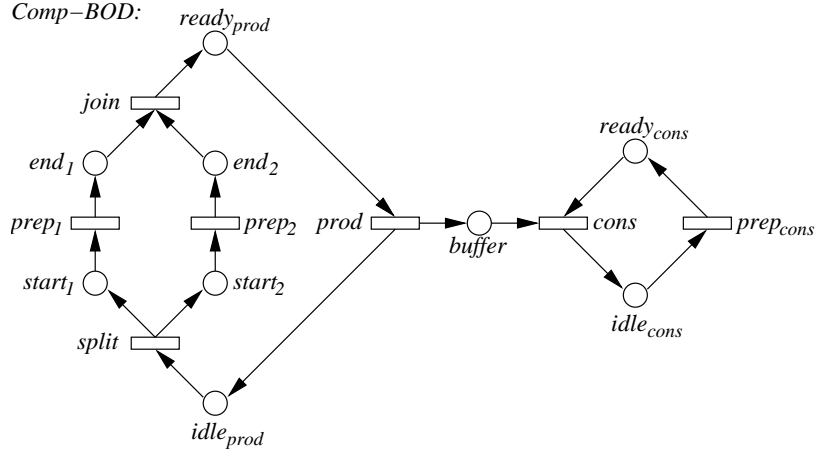


Fig. 15. Body of *Comp-COMP*

The import interfaces of the composition *Comp-COMP* are *PC-IMP*₂, *Pr-IMP*₁ and *Pr-IMP*₂. The import inclusion $imp_{Comp,C}: PC-IMP_2 \hookrightarrow Comp-BOD$ is given by the extension of $imp_{PC,2}$ along the transformation $xconnect'$, while $imp_{Comp,i}: Pr-IMP_i \hookrightarrow Comp-BOD$ for $i = 1, 2$ are given by $imp_{Comp,i} = imp'_{PC,1} \circ imp_{Pr,i}$.

6 Further Extensions

In this section we briefly discuss two further extensions of our component framework. In the first subsection we discuss how to construct a union operation of components with shared subcomponents. In this case we assume that our transformations in the general framework are HLR-transformations in the sense of Section 3. In the second subsection we discuss the compatibility of transformations of specifications with a suitable operational semantics of these specifications. In this case we assume that our specifications are operational specifications $TSP = (\Sigma, T)$, consisting of a signature Σ and a set T of Σ -computation steps, defining an operational semantics in the sense of [5]. The transformations are HLR-transformations over a corresponding category of operational specifications. More precisely we consider HLR-refinements, a special kind of HLR-transformations preserving computation steps and sequences, and conservative inclusions, a special kind of TSP -morphisms, and show the extension property required in the general framework in this context.

6.1 Union of Components

In the following we discuss how to construct a union operation for components, similar to the union of algebraic specification modules in [2]. Given components $COMP_i$ ($i = 0, 1, 2$) as in Sections 2 and 3, where $COMP_0$ can

be considered as shared subcomponent of $COMP_1$ and $COMP_2$, the union $COMP_3$ will be constructed separately by pushouts for import, export and body, provided that our HLR-category **CAT** has pushouts.

Without loss of generality we assume that the export transformation $exp_i: EXP_i \Rightarrow BOD_i$ ($i = 0, 1, 2$) consists of a direct transformation via a production p_i only. In fact in the case of transformation sequences of length $n \geq 2$ we are able to apply the concurrency theorem in [1] leading to a direct transformation via a so called concurrent production.

Now we are able to define a component morphism $f: COMP \rightarrow COMP'$ of components with direct transformations exp and exp' via productions p and p' . Note that f consists not only of morphisms f_E , f_B and f_I as shown explicitly in Figure 16, but of further morphisms between the productions and the context objects of the direct transformations, such that the diagram in Figure 16 commutes, where (1) is a commutative double-cube with double pushouts corresponding to the direct derivations, where the double pushouts are top and bottom of the double-cube.

$$\begin{array}{ccccc}
 EXP & \xrightarrow{exp} & BOD & \xleftarrow{imp} & IMP \\
 f_E \downarrow & & (1) \quad f_B \downarrow & & (2) \quad f_I \downarrow \\
 EXP' & \xrightarrow{exp'} & BOD' & \xleftarrow{imp'} & IMP'
 \end{array}$$

Fig. 16. Component morphism

This leads to a category **COMP** of components and component morphisms.

For given component morphisms $f_i: COMP_0 \rightarrow COMP_i$ ($i = 1, 2$) we are able to construct the union component $COMP_3$ with component morphisms $g_i: COMP_i \rightarrow COMP_3$ ($i = 1, 2$) by pushouts and amalgamated transformations in the given HLR-category **CAT** leading to Figure 17 in the category **COMP**. Note that $exp_3: EXP_3 \Rightarrow BOD_3$ in $COMP_3$ becomes a direct transformation via the amalgamated production $p_3 = p_1 +_{p_0} p_2$ using a theorem in analogy to distributivity of union (see e. g. Theorem 3.18 in [2]).

$$\begin{array}{ccc}
 & COMP_0 & \\
 f_1 \swarrow & & \searrow f_2 \\
 COMP_1 & (PO) & COMP_2 \\
 g_1 \searrow & & \swarrow g_2 \\
 & COMP_3 &
 \end{array}$$

Fig. 17. Union of components

For example the producer/consumer system from the last sections could be constructed as the union of a component describing the producer and a component describing the consumer, where the shared subpart of both com-

ponents would be the refinement of the *prodcons*-transition by the buffer and the transitions *prod* and *cons*.

6.2 Compatibility of High-Level Transformations and Operational Semantics

In this subsection we sketch how to establish the compatibility of our transformation semantics and an abstract form of an operational semantics. Following [5,4], the operational semantics of a specification is defined by means of algebra transformations, called computation steps below in order to avoid confusion with our notion of transformations in previous sections. In particular, in [5] it is explicitly shown how Petri Nets and the single and double-pushout approaches for graph transformation can be seen as special cases. Nevertheless, it remains open how the specific concepts introduced below are applicable in these areas. Hence, we consider systems where states are many-sorted algebras and where computation steps are seen as pairs of algebras together with a tracking map, which is a partial injective function identifying the elements of A which have not been removed after the transformation. Then, we define the kind of inclusions that can be used as import connections, which are inclusions where the body rules cannot transform the import part of a state. We also provide a definition of refinement, which are high-level transformations that preserve computations. Finally, we present the main result that shows the satisfaction of the extension property in this framework. This result should be the basis for proving the full compatibility of the operational semantics of components and their composition with the given transformation semantics.

As said above, we consider systems where states are many-sorted algebras and where Σ -computation steps, $\tau = (A, B, f: A \rightarrow B)$, consist of a pair of Σ -algebras and a tracking map, which is a partial injective function. Then, an operational specification TSP is a pair, (Σ, T) , consisting of a signature and a set of Σ -computation steps.

Given a signature morphism $h: \Sigma_1 \rightarrow \Sigma_2$ and a Σ_1 -computation step τ , we denote by $h^*(\tau)$ the set of Σ_2 -computation steps (A', B', f') , where $A = U_h(A')$ and $B = U_h(B')$. In addition, in the rest of the signature, B' and A' coincide, which means that f' can be seen as f extended by the identity mapping in all sorts not in $h(\Sigma_1)$. Similarly, given a set of Σ_1 -computation steps T , we denote by $h^*(T)$ the set $\bigcup_{\tau \in T} h^*(\tau)$.

Operational specifications form a category where a morphism $h: TSP_1 \rightarrow TSP_2$ is a signature morphism $h: \Sigma_1 \rightarrow \Sigma_2$ such that $h^*(T_1) \subseteq T_2$.

Conservative inclusions, $i: TSP_1 \hookrightarrow TSP_2$ in our category of specifications are signature inclusions such that for every computation step $\tau = (A, B, f)$ in $T_2 \setminus T_1$ we have $U_i(A) = U_i(B)$, which means that the computation step can not modify the included part of a given state.

We say that an HLR-transformation $trafo: TSP_1 \Rightarrow TSP_2$ defined by means of the application of a rule $r = (L \leftarrow K \rightarrow R)$ from an operational specification TSP_1 to TSP_2 is an HLR-refinement if:

- (i) $\Sigma_L = \Sigma_K$, i.e. rules are non-deleting on the signature part.
- (ii) If in TSP_1 there is a computation leading from the Σ_1 -algebra A_1 to A'_1 using computation steps from T_1 then for every Σ_2 -algebra A_2 , such that $A_1 = U_h(A_2)$, there should exist a computation in TSP_2 leading from A_2 into another Σ_2 -algebra A'_2 such that $A'_1 = U_h(A'_2)$ using computation steps from T_2 , where $h: \Sigma_1 \rightarrow \Sigma_2$ is the signature morphism induced by the transformation. This means that an HLR-refinement preserves the computations in TSP_1 along the associated HLR-transformation.

In this context, the extension property can be proved: Given operational specifications TSP_1 , TSP_2 and TSP'_1 , if the HLR-transformation $trafo: TSP_1 \Rightarrow TSP_2$, by means of a rule r via a morphism h , is an HLR-refinement, and if $i_1: TSP_1 \hookrightarrow TSP'_1$ is a conservative inclusion, then r can be applied to TSP'_1 via $i_1 \circ h$ leading to TSP'_2 . Moreover, the transformation $trafo': TSP'_1 \Rightarrow TSP'_2$ defined by this application is an HLR-refinement and $i_2: TSP_2 \hookrightarrow TSP'_2$ is a conservative inclusion.

7 Conclusion

In this contribution we have shown how the generic component approach in our FASE-paper [3] can be made more explicit. In fact we have instantiated the generic transformations in this approach by transformations of high-level replacement systems, a well-known abstract concept in the theory of graph transformations. We have explicitly shown an example based on low-level Petri nets, while a high-level net example for composition of components was given in [3]. It is ongoing work to study instantiations of our generic component approach in more detail for different kinds of graph- and net-based as well as other visual modeling techniques.

Acknowledgement

We would like to thank the referees for their valuable comments leading to an improved version of the paper.

This work is partially supported by the German DFG project IOSIP, by the German BMBF project on “Continuous Software Engineering”, by the Spanish project MAVERISH (TIC2001-2476-C03-01) and by the CIRIT Grup de Recerca Consolidat 2001SGR 00254.

References

- [1] Ehrig, H., A. Habel, H.-J. Kreowski, and F. Parisi-Presicce, *Parallelism and Concurrency in High-Level Replacement Systems*, Math. Struct. in Comp. Science **1** (1991), Cambridge Univ. Press, pp. 361–404.

- [2] Ehrig, H., and B. Mahr, “Fundamentals of Algebraic Specification 2: Module Specifications and Constraints”, vol. 21 of EATCS Monographs on Theor. Comp. Science, Springer Verlag, Berlin (1990).
- [3] Ehrig, H., F. Orejas, B. Braatz, M. Klein, and M. Piirainen, *A Generic Component Framework for System Modeling*, Proc. FASE 2002, Springer LNCS 2306 (2002), pp. 33–48.
- [4] Große-Rhode, M., *Integrating Semantics for Object-Oriented System Models*, Proc. ICALP 2001, Springer LNCS 2076 (2001), pp. 238–255.
- [5] Orejas, F., H. Ehrig, and E. Pino, *Tight and Loose Semantics for Transformation Systems*, Recent Trends in Algebraic Development Techniques, Springer LNCS 2267 (2001), pp. 238–255.
- [6] Padberg, J., H. Ehrig, and L. Ribeiro, *Algebraic High-Level Net Transformation Systems*, Math. Struct. in Comp. Science **5** (1995), Cambridge Univ. Press, pp. 217–256.
- [7] Padberg, J., and G. Taentzer, *Embedding of Derivations in High-Level Replacement Systems*, Technical Report, TU Berlin, 1993.