Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 108 (2004) 53–67
www.elsevier.com/locate/entcs

A Generic Framework for Connector Architectures based on Components and Transformations

H. Ehrig, J. Padberg, B. Braatz, M. Klein[1]

Institute for Communication and Software Technology
Technical University Berlin, Germany

F. Orejas, S. Perez, E. Pino[2]

Departament de Llenguatges i Sistemes Informàtics,
Universitat Politécnica de Catalunya, Barcelona, Spain

Abstract

The intention of this paper is to extend our generic component framework presented at FASE 2002 [4] to a specific kind of connector architectures similar to architectural connections in the sense of Allen and Garlan [1]. In our generic component framework we have considered components with explicit import, export and body parts connected by embeddings and transformations and composition of components with a compositional transformation semantics. Our framework, however, was restricted to components with a single import and export interface. Here we study architectures based on connectors with multiple imports and components with multiple exports. Architectures studied in this paper are built up from components and connectors in a noncircular way. The semantics of an architecture is defined by reduction step sequences in the sense of graph reductions. The main result shows existence and uniqueness of the semantics of an architecture as a normal form of reduction step sequences. Our generic framework is instantiated on one hand to connector architectures based on CSP as the formal specification technique in the approach by Allen and Garlan. On the other hand it is instantiated to connector architectures based on high-level-replacement systems in general and Petri nets in particular. A running example using Petri nets as modeling technique illustrates all concepts and results.

Keywords: Components, connector architecture, Petri net transformations

[1] Email:{ehrig,padberg,bbraatz,klein}@cs.tu-berlin.de
[2] Email:{orejas,sperezl,pino}@lsi.upc.es

doi:10.1016/j.entcs.2004.01.012

1 Introduction

The importance of architecture descriptions has become most obvious over the last decade (see e.g. [17,18,7,10,6]). Various formalisms have been proposed to deal with the complexity of large software systems. In order to build up large software systems from smaller parts, a flexible component concept for software systems and infrastructures are a useful and widely accepted abstraction mechanism (see e.g. [19,12,8]). Although there are many approaches available, only few are general enough to be used for different specification techniques. To achieve a generic concept the focus has to be on the fundamental issues of components and component-based systems. These are the interfaces, the compositionality of components and its embedding into the environment.

In our FASE 2002 paper [4] we have presented a generic component framework for system modeling that can be used for a large class of semi-formal and formal modeling techniques. According to this concept a component consists of a body, an import, and an export interface, and connections between import and body as well as export and body. These connections are again generic to allow a great variety of instantiations. We only require having suitable notions of embeddings, called inclusions in [4], and transformations (e.g. refinements) between specifications, such that the import connection of a component defines an embedding and the export connection a transformation. The connection between import and export interfaces of different components are also represented by transformations. In fact, 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 [5,2,16]. In this paper we extend our generic component framework discussed above to a specific kind of connector architectures motivated by architectural connections in the sense of Allen and Garlan [1]. In fact, our generic framework in [4] is restricted to components with a single import and export interface. We consider architectures using connectors with multiple imports and components with multiple exports that allow connecting one connector to several different components. The key concept for the corresponding composition of a connector with components is a parallel extension diagram for transformations. This generalizes the notion of an extension diagram in [4] which is the key concept to define transformation semantics for components and to prove compositionality. Architectures studied in this paper are built up from components and connectors in a noncircular way. The semantics of an architecture is defined by reduction step sequences, where in each reduction step one connector is composed with all adjacent components. The main result shows existence and uniqueness of the semantics as a normal form of reduction step sequences.

This paper continues with the construction of the connector architecture in Section 2. In this and the following sections we have an ongoing example using Petri nets. In Section 3 we show the composition of components with connectors. Subsequently we show the existence of a unique semantics for architectures in Section 4. In Section 5 we give a concrete instantiation to CSP and a more abstract instantiation to high-level replacement systems including Petri nets as a concrete case. In Section 6 we conclude with a brief discussion of related work and an outlook to future research.

2 Construction of Connector Architectures

In this section we present the main syntactical concepts of our general framework for connector architectures. The framework in [4] as well as the framework in this paper is generic not only concerning the underlying concept of semi-formal or formal specifications, but also concerning the concept of transformations in order to model abstraction and refinement between interfaces and body of one component, or between import and export interfaces between different components. In this section we only require that transformations are closed under composition and that we have a special kind of transformations, called embeddings, which intuitively model inclusion of specifications, and a notion of independence of embeddings explained below. Motivated by architectural connections in the sense of Allan and Garlan [1] we distinguish in this paper components and connectors with multiple interfaces, while we have considered only components with single interfaces in [4]. Now a component $COMP = (B, e_1 : E_1 \Longrightarrow B, ..., e_n : E_n \Longrightarrow B)$ for $n \geq 0$ is given by the body B and a family of export interfaces E_i with export transformations $e_i : E_i \Longrightarrow B$ for $i \in \{1, ..., n\}$. A connector $CON = (B, b_1 : I_1 \to B, ..., b_n : I_n \to B)$ for $n \geq 2$ is given by the body B and a family of import interfaces I_i with body embeddings $b_i : I_i \to B$ for $i \in \{1, ..., n\}$. We assume that the family of embeddings $b_i : I_i \to B$ for each connector is **independent**. This means intuitively that the import interfaces I_i of B are pairwise disjoint. Now we can define formally how a connector connects different components. Given a connector $CON = (B, b_1, ..., b_n)$ of arity n, and n components $COMP_i = (B_i, e_{i_1}, ..., e_{i_{m_i}})$ of arity m_i with connector transformations $con_i : I_i \Longrightarrow E_{i_k}$ with $1 \leq k \leq m_i$ for $i \in \{1, ..., n\}$ then we obtain the **connector diagram** in Figure 1 and the **connector graph** in Figure 2:

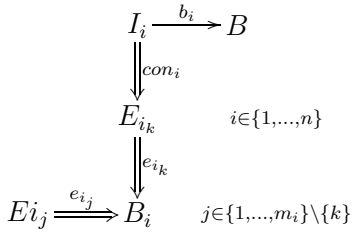

Fig. 1. Connector Diagram

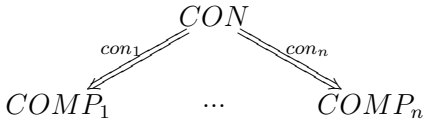

Fig. 2. Connector graph

Remarks:

- A connector diagram consists of n import interface nodes I_i of $n+1$ body nodes B_i and B, and of $\sum_{i=1}^{n} m_i$ export interface nodes E_{i_j}, even if some of the specifications may be equal, e.g. $B_1 = B_2$
- Circular connections as in (2) are forbidden, unless we duplicate the body as in (1) of Figure 3. Otherwise the semantics of such a circular architecture is not defined, as it would cause the identification of the export interfaces E_{1_1} and E_{1_2} of component $COMP_1$ or other kinds of unwanted side effects.

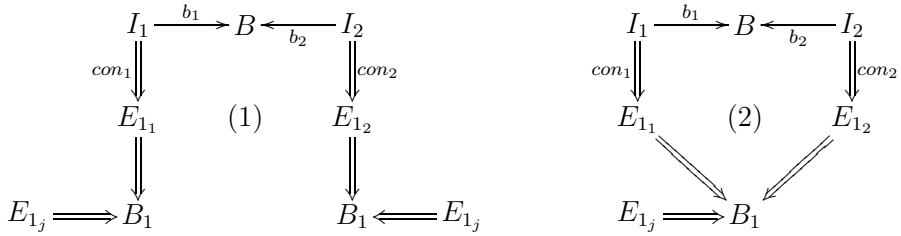

Fig. 3. Noncircular and circular connector diagrams

- A well-known result from graph theory for undirected graphs (i.e. forgetting the direction of the arcs) is that in a connected, noncircular graph the number of nodes exceeds the number of edges by one: $\#nodes(G) = \#edges(G) + 1$. In Figure 3 we have $\#nodes(1) = 9 = \#edges(1) + 1$, but $\#nodes(2) = 7 \neq 8 = \#edges(2) + 1$

Now, we introduce the architecture based on connectors and components. Similarly to connectors we obtain an architecture diagram and an architecture graph. The first describes the architecture at the level of specifications and the second as a graph, where nodes are connectors or components.

An **architecture** A of arity (k, l) consists of k components and l connectors, an architecture diagram D_A and an architecture graph G_A: The **architecture diagram** D_A is a diagram built up from the l connector diagrams and the connection transformations satisfying the following conditions

(i) **Connector Condition:** Each import interface I of a connector is connected by an arrow, labeled with a connection transformation $con : I \Longrightarrow$

E, to exactly one export interface E of one component.

(ii) **Component Condition:** Each export interface E of a component is connected at most to one import interface I of a connector by an arrow from I to E, labeled with a connection transformation $con : I \Longrightarrow E$.

(iii) **Noncircularity:** The architecture diagram D_A is connected and noncircular aside from the arrows' direction.

The **architecture graph** G_A of architecture A is obtained from the architecture diagram D_A by shrinking each connector diagram in D_A to the corresponding connector graph. Hence, it consists of nodes labeled by the connectors and components and arrows in between labeled with the corresponding connection transformations.

Petri Net Example for a Connector Architecture: Preparing a Party

To provide a concrete example for a connector architecture we have chosen place/transition nets as the underlying specification technique. In Section 5 we discuss the instantiation of connector architectures to Petri nets. The main focus of this simple and small example is to illustrate the introduced concepts, using a scenario from everybody's life: preparing a party. In order to show the Petri nets explicitly, a component is drawn as a rectangular, including one or more rectangulars in the first row, where each contains one export net. The rectangular in the second row contains the body net and the component name. A connector is drawn as a rectangular as well. In the first row we have exactly one rectangular, containing the body. The second row comprises at least two rectangulars, each containing one import net.

The components comprise the following activities: **Comp_invite** in Figure 4 the invitation of guests and the management of cancellations. The component has merely one export interface **Ei**. The places of the export net are preserved under the transformation, but the transition is replaced by the whole subnet in between the places. In Figure 5 we give the usual diagram for the component **Comp_invite**, but without the explicit Petri nets.

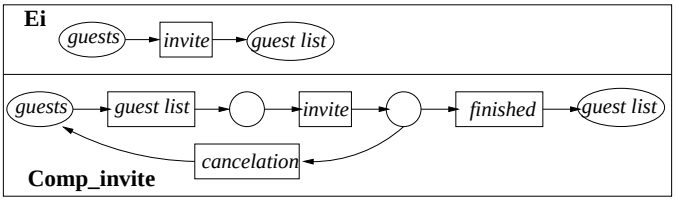

Fig. 4. Component **Comp_invite**

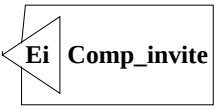

Fig. 5. Classical Diagram

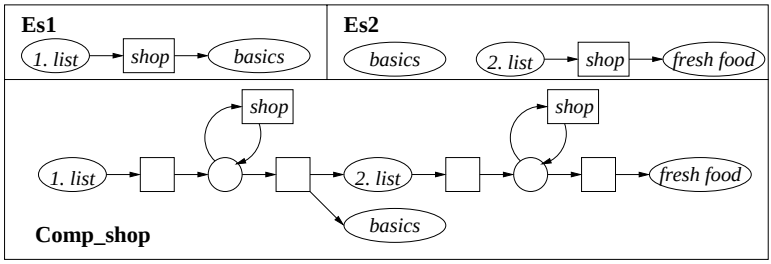

Fig. 6. Component **Comp_shop**

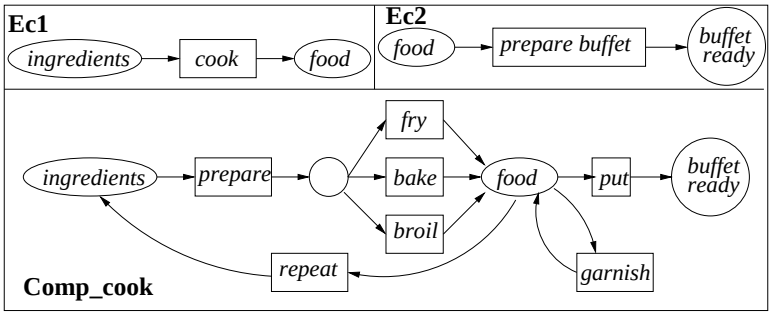

Fig. 7. Component **Comp_cook**

The component **Comp_shop** models the shopping, where we assume that the party requires shopping two times sequentially, once for the basics (beverages, potato crisps, olives, etc.) and once for the fresh food (bread, cheese, fruits, etc.). So we have two export nets **Es1**, **Es2** that are transformed into the body net. These transformations keep the places and replace the transition by the subnet in between. The last component **Comp_cook** comprises the cooking and preparations of the buffet, and provides therefore two export interfaces **Ec1** and **Ec2**.

The connector **Con_week** in Figure 8 connects the activities that concern the preparations of the week before the party and provides two import nets **Iw1** and **Iw2**. These are mapped by inclusion into the body net. In Figure 9 we give the usual diagram for the connector **Con_week**, but without the nets.

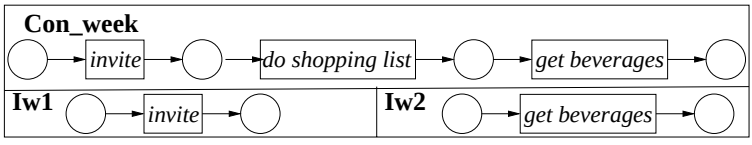

Fig. 8. Connector **Con_week**

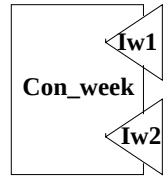

Fig. 9. Classical Diagram

The connector **Con_day** in Figure 10 connects the preparations on the day of the party, and provides two imports as well.

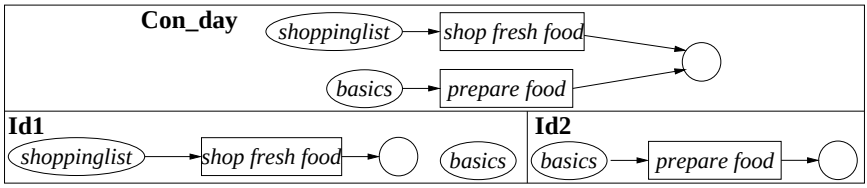

Fig. 10. Connector **Con_day**

The architecture diagram D_A is given in Figure 11, where the transformations between the import nets and the corresponding export nets merely rename places or transitions. In Figure 11 we use the names of connectors and components for the corresponding bodies.

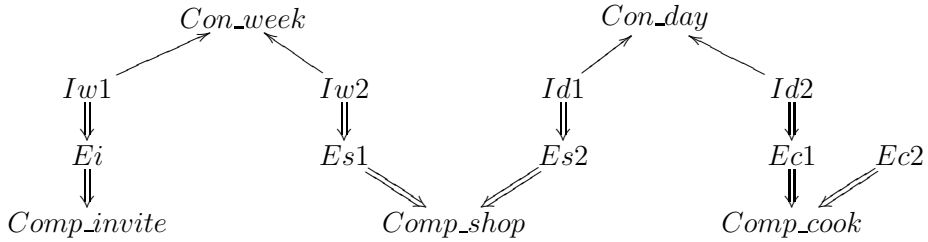

Fig. 11. Architecture diagram D_A

3 Composition of Components

In this section we present our concept for the composition of components by a connector, which is the basic step for the construction of the semantics of architectures in the next section. For this purpose we have to require in our generic framework an essential property for embeddings and transformations: The **parallel extension property** is a key concept which generalizes the extension property in [4] from single (sequential) to multiple (parallel) transformations. The intuitive idea is that each family of transformations $(t_i : SPEC_i \Rightarrow SPEC_i')_{i\in I}$ can be extended to a parallel transformation $t : SPEC \Rightarrow SPEC'$, provided that we have independent embeddings $b_i : SPEC_i \Rightarrow SPEC'$. More precisely, the **parallel extension property** means the following: For each family $(b_i : SPEC_i \to SPEC)_{i\in I}$ of independent embeddings and for each family of transformations $(t_i : SPEC_i \Longrightarrow SPEC_i')_{i\in I}$ there is a canonical (parallel) transformation $t : SPEC \Longrightarrow SPEC'$ together with the independent embeddings $(b_i' : SPEC_i' \to SPEC)_{i\in I}$ leading to the **parallel extension diagram** (1) in Figure 12. Moreover, parallel extension diagrams are required to be closed

$$\begin{array}{ccc} SPEC_i & \xrightarrow{b_i} & SPEC \\ {\scriptstyle t_i}\Big\Downarrow & (1) & \Big\Downarrow{\scriptstyle t} \\ SPEC_i' & \xrightarrow{b_i'} & SPEC' \end{array}$$

Fig. 12. Parallel extension diagram

under **vertical composition** and to **include the (classical) extension diagram** in the sense of [4] as a special case where all but one of the transformations t_i are identical transformations. Now we are able to define the composition of components.

The **composition of n components by a connector of arity n** is defined as follows: Given the corresponding connector diagram (see Figure 1) we construct the corresponding parallel extension diagram (1) in Figure 13. The result of the composition of the components $COMP_1, ..., COMP_n$ by the connector CON with the connection transformations $con_1, ..., con_n$ is again a component $COMP = (B', (e'_{ij} : E_{ij} \Longrightarrow B')_{ij \in I \times J})$ with $e'_{ij} := b'_i \circ e_{ij} : E_{ij} \Longrightarrow B'$ for each $ij \in \{1, ..., n\} \times (\{1, ..., m_i\} \setminus \{k\}) = I \times J$. In this case we say that e'_{i_j} are extensions of e_{i_j}.

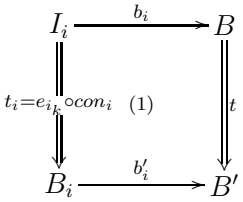

Fig. 13. Composition

In case of binary components and binary connectors we use the following nice infix notation $COMP = COMP_1 +_{CON} COMP_2$. Otherwise we use the notation $COMP = CON(COMP_1, ..., COMP_n, con_1, ...con_n)$. The result below can be extended to the composition of connectors with multiple import interfaces.

Theorem 3.1 (Associativity of Binary Component Composition) *Given an architecture A with binary components and binary connectors with the architecture graph G_A in Figure 14, then we have the following associativity law:*

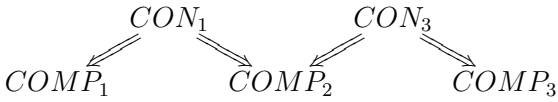

Fig. 14. Binary connectors and components

$$(COMP_1 +_{CON_1} COMP_2) +_{CON_2} COMP_3 = COMP_1 +_{CON_1} (COMP_2 +_{CON_2} COMP_3)$$

Proof idea: Each side of the equation can be shown to be equal to a parallel composition of $COMP_1$, $COMP_2$, and $COMP_3$ via CON_1 and CON_2 using the parallel extension property.

Composition for the Petri Net Example

Composing first the components **Comp_invite** and **Comp_shop** using the connector **Con_week** we obtain the new component **Comp_invshop** illustrated in Figure 15.

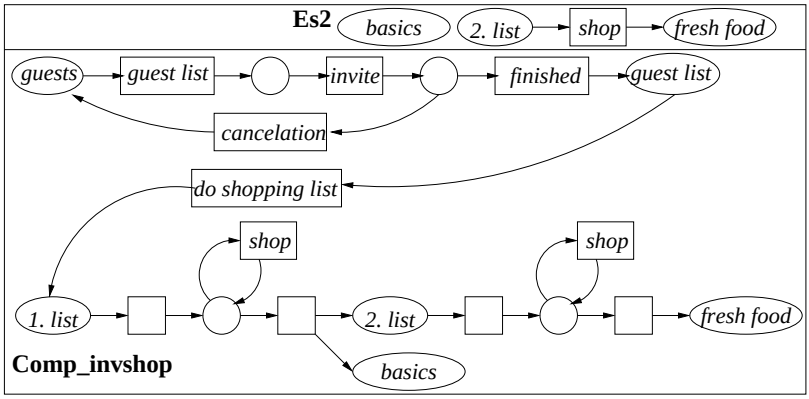

Fig. 15. **Comp_invshop**

The corresponding parallel extension diagram for the construction of the body B' of **Comp_invshop** is depicted in Figure 16, where we again use the names of components or connectors also for their bodies.

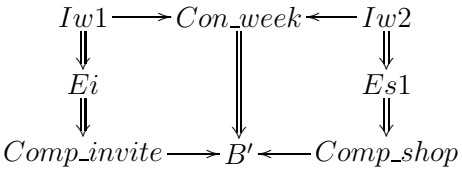

Fig. 16. Parallel Extension diagram

Connecting this component via the connector **Con_day** with the components **Comp_cook** we obtain the component **Comp_invshopcook** in Figure 17. But we can also change the order in which we compose.

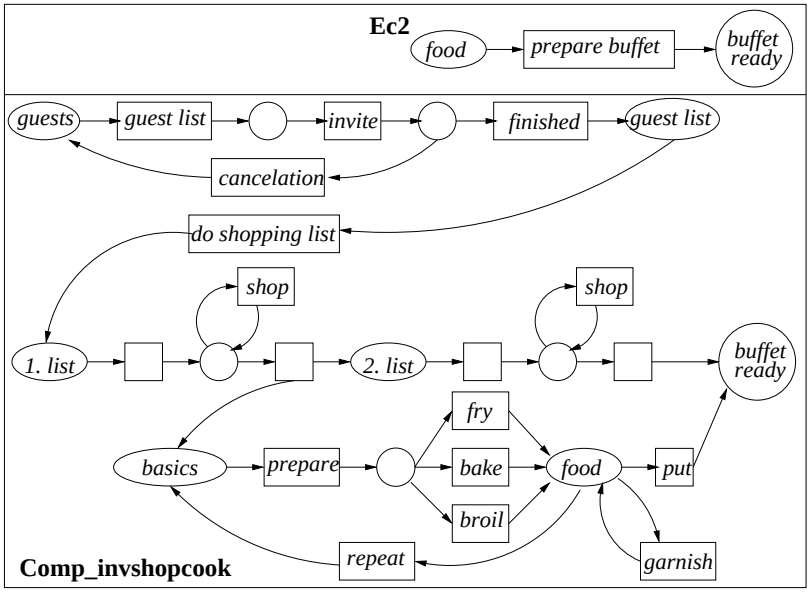

Fig. 17. **Comp_invshopcook**

So, using the connector **Con_day** we first compose the components **Comp_shop** and **Comp_cook**. Connecting this one via the connector **Con_week** with the component **Comp_invite** we again obtain the component **Comp_invshopcook** in Figure 17. This commutativity is ensured by Theorem 3.1.

4 Semantics of Architectures

In this section we define the semantics of architectures. In fact, we construct a well-defined single component as semantics, which corresponds to the composition of all components using all connectors of the given architecture. More precisely, for an architecture there are **reduction rules** that visualize step by step the composition of components via connectors. Both reduction rules are productions $p = (L \leftarrow K \rightarrow R)$ in the sense of the algebraic approach to graph transformation [5]. A derivation step in this approach is given by two pushout diagrams (1) and (2) in Figure 18, written $G \Longrightarrow H$ via (p, m), where $m : L \rightarrow G$ is a graph morphism, that represents the match of L in G. Intuitively, we remove $(L - K)$ from G in step 1 leading to the context graph D in (1). And then we add $(R - K)$ leading to the result H in (2). The pushout property of (1) and (2) means intuitively that G is the gluing of D and L along K in (1), respectively H is the gluing of D and R along K in (2).

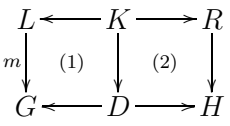

Fig. 18. $G \Longrightarrow H$

Given an architecture A with the architecture diagram D_A and the architecture graph G_A there is for each connector CON the following **diagram reduction rule** CON_D:

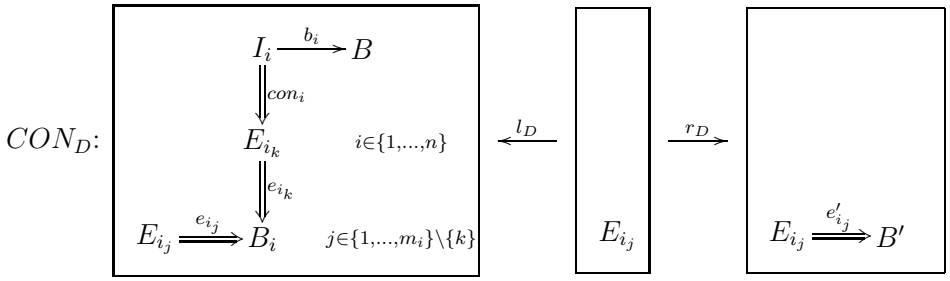

Fig. 19. Diagram reduction rule

where B' and $e'_{i_j} = e_{i_j} \circ b'_i$ is defined by the composition:

$$COMP = CON(COMP_1, ..., COMP_n, con_1, ...con_n)$$
$$= (B', (e'_{ij} : E_{ij} \Longrightarrow B')_{ij \in I \times J})$$
$$\text{for each } ij \in \{1, ..., n\} \times (\{1, ..., m_i\} \setminus \{k\}) = I \times J.$$

The corresponding **graph reduction rule** CON_G is given in Figure 20 where $COMP_1, ..., COMP_n$ are mapped to $COMP$. So, a reduction step $CON_D : D_A \Longrightarrow D_{A'}$, respectively $CON_G : G_A \Longrightarrow G_{A'}$ is given by a derivation step at the level of architecture diagrams, respectively architecture graphs.

For both derivation steps we have inclusions for the matches. Note that r_G is neither injective nor label-preserving, nevertheless for the reduction rule $CON_G : G_A \Longrightarrow G_{A'}$ the labels of G'_A are well-defined by G_A and $COMP$.

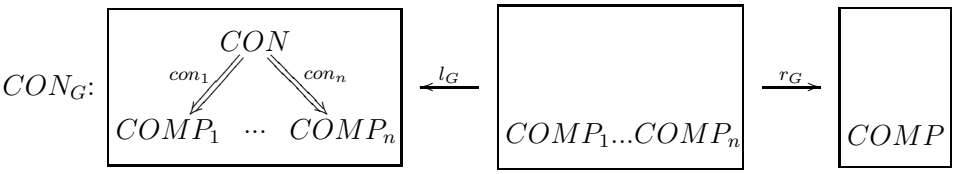

Fig. 20. Graph reduction rule

We can show by an **Architecture Reduction Lemma** that an **architecture reduction rule** $CON = (CON_D, CON_G)$ reduces an architecture A to a well-defined smaller architecture A' with $D_{A'}$ and $G_{A'}$ as defined above. The application of CON is denoted by $A \overset{CON}{\Longrightarrow} A'$. A' is smaller than A in the following sense: If A is of arity (k, l) we can show that A' is of arity $(k - n + 1, l - 1)$, if CON has arity n. This means: Given an architecture A consisting of k components and l connectors we obtain the architecture A' with $k - n + 1$ components and $l - 1$ connectors.

Now we can give the **semantics of an architecture** as the result of as many reduction rules as possible. The semantics of an architecture A is any component $COMP$ obtained by a sequence of architecture reduction steps $A \overset{*}{\Longrightarrow} COMP$ from A to $COMP$. The main result given in Theorem 4.1 shows that this semantics always exists and is unique. The main reason therefore is given by the Church-Rosser property, stating that any two architecture reduction steps are locally confluent.

Theorem 4.1 (Existence and uniqueness of architecture semantics) *For each architecture A there is a unique component $COMP$ which is the semantics of A. $COMP$ is obtained by* **any** *reduction sequence, where connectors of A are reduced in arbitrary order:*

$$A \overset{*}{\Longrightarrow} COMP$$

Proof idea: The architecture reduction step sequences have unique normal forms, as they satisfy the Church-Rosser property due to the following reasoning: If the matches of CON_1 and CON_2 in the architecture diagram D_{A_0} of A_0 are disjoint, then they are independent and we obtain the result by the Church-Rosser Theorem for graph transformations. Otherwise, due to non-circularity of D_{A_0}, the matches can overlap at most in one component which allows applying Theorem 3.1, respectively an extension with multiple import

interfaces. So, each maximal sequence has length n, where n is the number of connectors in A.

Semantics of the Petri Net Example

The application of the reduction rule $D_A \overset{CON_{WEEK}}{\Longrightarrow} D_{A_1}$ that eliminates the connector **Con_week** results in the architecture diagram D_{A_1} given in Figure 21. There we have the new component **Comp_invshop** with one export **Es2**, that is connected as before to the import **Id1** of connector **Con_day**. The application of $D_{A_1} \overset{CON_{DAY}}{\Longrightarrow} D_{A_3}$ that eliminates the connector **Con_day** yields then the semantics of A. This is the component **Comp_invshopcook** in Figure 17 with only one export left, namely **Ec2**. Theorem 4.1 ensures that this semantics exists uniquely. If we start for example with the application of $D_A \overset{CON_{DAY}}{\Longrightarrow} D_{A_2}$ that eliminates the connector **Con_day** we obtain a component **Comp_shopcook** with the two exports **Es1** and **Ec2**. The corresponding architecture diagram D_{A_2} is given in Figure 22. The subsequent application of $D_{A_2} \overset{CON_{WEEK}}{\Longrightarrow} D_{A_3}$ eliminating the connector **Con_day** yields then again the component **Comp_invshopcook**.

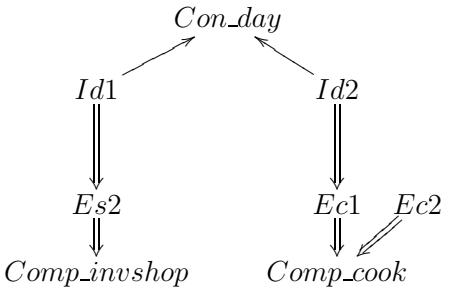

Fig. 21. Architecture Diagram D_{A_1}

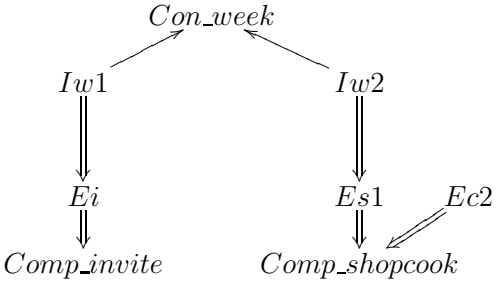

Fig. 22. Architecture Diagram D_{A_2}

5 Instantiations

In [1] Allen and Garlan introduced architectural connectors using CSP [9] as specification formalism. In this section we shall very briefly sketch how CSP and Petri nets fit into our transformation framework.

CSP can be seen as an instance of our transformation framework as follows. First, we consider that a CSP specification $P = (\Sigma, Exp)$ where Σ is a *process signature* (the set of symbols that can be used in P) and Exp is a CSP process expression built over symbols in Σ. Then, we consider that a process P is embedded in a process Q if P is a parallel component of Q. More precisely, $P_1 = (\Sigma_1, Exp_1)$ is **embedded** in $P_2 = (\Sigma_2, Exp_2)$ if $\Sigma_1 \subseteq \Sigma_2$) and $Exp_2 \equiv Exp_1|Exp'$ for some process expression Exp'. P_1 and P_2 are independently embedded in P_3 iff $Exp_3 \equiv Exp_1|Exp_2|Exp'$ for some process expression Exp'.

Finally, we consider that transformations are just CSP refinements modulo a signature embedding. This means that a **CSP transformation** $t\colon P_1 = (\Sigma_1, Exp_1) \Rightarrow P_2 = (\Sigma_2, Exp_2)$ is an injective mapping $t\colon \Sigma_1 \to \Sigma_2$ such that the translation of Exp_1 through t satisfies that the failures and divergences of Exp_2 are, respectively, a subset of the failures and divergences of $t(Exp_1)$, where $t(Exp_1)$ denotes the translation of Exp_1 by the renaming of events defined by t.

Then, CSP can be seen as an instance of our generic approach as a consequence of the **existence of parallel extensions**:

If, for $i = 1, 2$, $t_i\colon P_i = (\Sigma_i, Exp_i) \Rightarrow P_i' = (\Sigma_i', Exp_i')$ are transformations and P_1, P_2 are independently embedded in $P_3 = (\Sigma_3, Exp_3)$, then $t_3\colon P_3' = (\Sigma_3', Exp_3')$ is a parallel extension of t_1 and t_2, where $\Sigma_3' = \Sigma_3 + (\Sigma_1' - t_1(\Sigma_1)) + (\Sigma_2' - t_2(\Sigma_2))$, where $+$ and $-$ denote disjoint union and set subtraction, $t_3\colon \Sigma_3 \to \Sigma_3'$ is the identity and Exp_3' is the process expression obtained by substituting Exp_1 by Exp_1' and Exp_2 by Exp_2' in Exp_3.

High-level replacement systems have been introduced in [2] as a generalization of the *Double Pushout* approach for graph transformations from graphs to several kinds of high-level structures in suitable categories including a large variety of different kinds of graphs and Petri nets ([4]). A transformation in the framework of this paper corresponds to a derivation sequence of high-level structures (e.g. Petri nets). The extension property for transformations considered in our general framework for components [4] is well-known as the Embedding Theorem in the theory of high-level replacement systems that holds if a consistency condition between embeddings and transformations is required. A similar consistency condition has to be formulated for the parallel extension property considered in this paper. This property can be shown for high-level replacement systems by considering the Embedding Theorem and the Parallelism Theorem shown in [2]. Moreover, we may allow also some overlapping of the embeddings, provided that the transformations preserve the overlapping part in the sense of parallel independence. This situation occurs for the embeddings of the connector **Con_day** in Figure 10 in our running example.

6 Conclusion

In this paper we have presented a general framework for connector architectures, based on our generic framework for components in [4], and motivated by the architectural connectors approach of Allen and Garlan in [1] which is shown to be a specific instance. It may be mentioned that the papers

[11,20] concerning architecture reconfiguration based on graph transformation techniques can be considered complementary to ours, because graph transformations and high-level replacement play already a fundamental role in our approach.

Future work comprises further and more concrete instantiations especially high-level Petri nets and graph transformation systems. Moreover, larger case studies to evaluate the practical impact are an important task. On the theoretical side the instantiations to general high-level replacement systems has to be worked out in more detail and our framework should be extended to other aspects studied in [4] and architecture reconfiguration in [11,20].

Acknowledgement

This work is partially supported by the TMR network SEGRAVIS and the Spanish project MAVERISH (TIC2001-2476-C03-01) and by the CIRIT Grup de Recerca Consolidat 2001SGR 00254.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.

[2] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1990.

[4] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A generic component concept for system modeling. In *Proc. FASE '02*, LNCS 2306. Springer, 2002.

[5] H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[6] D. Garlan, R. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proc. of CASCON'97*, pages 169–183, 1997. http://www.cas.ibm.ca/cascon/cfp.html.

[7] F. Griffel. *Componentware – Konzepte und Techniken eines Softwareparadigmas*. dpunkt Verlag, 1998.

[8] V. Gruhn and A. Thiel. *Komponentenmodelle: DCOM, JavaBeans, EnterpriseJavaBeans, CORBA*. Addison-Wesley, 2000.

[9] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] C. Hofmeister, R. Nord, and D. Soni. *Describing Software Architecture in UML*, pages 145–159. Kluwer Academic Publishers, 1999.

[11] M. Löwe. Evolution pattern. postdoctoral thesis, Technical University of Berlin, 1997.

[12] S. Mann, B. Borusan, H. Ehrig, M. Große-Rhode, R. Mackenthun, A. Sünbül, and H. Weber. Towards a component concept for continuous software engineering. Technical Report 55/00, FhG-ISST, 2000.

[13] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement.* PhD thesis, Technical University Berlin, 1996. Shaker Verlag.

[14] J. Padberg. Petri net modules. *Journal on Integrated Design and Process Technology*, 6(4):105–120, 2002.

[15] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.

[16] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[17] M. Shaw, R. Deline, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–315, 1995.

[18] M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[19] C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[20] M. Wermelinger and J. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.