Towards Multiple Access in Generic Component Architectures [1]

M. Klein, J. Padberg [2]

Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany

F. Orejas [3]

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

Abstract

The paper introduces an abstract framework for the specification of components with multiple require and provide interfaces that allows the specification of multiple access to a single provide interface. This framework can be regarded as a generalization of abstract hierarchical and connector-based component specification approaches. The main ideas are clarified in a sample specification, a component architecture for a web browser suite. For this, elementary nets are applied and are shown to be an instantiation of the abstract framework.

Key words: Component Architectures, Reduction Semantics

1 Introduction

By now component-based software development is becoming nearly a standard in large scale software engineering (see e.g. [21,22,33,34]), for several reasons: For example, components implemented once can easily be integrated in other projects requiring the same functionalities. It is possible to buy components with explicitly defined interfaces, thus, time pressure in the development of software projects can be relaxed by paying for a piece of code that is quickly integrated into the project. Changes of a component body, or even a full exchange, can be processed encapsulated, i.e. with no effects for the component's environment as long as the corresponding interfaces are preserved. See [24] for a survey of component-based software engineering.

[1] **Acknowledgment:** This work is partially supported by the TMR network SEGRAVIS.
[2] Email: [klein, padberg]@cs.tu-berlin.de
[3] Email: orejas@lsi.upc.edu

But despite the wide acceptance of component-based software development approaches, there is still a lack of specification techniques suited for component-based design. Especially the application of formal specification techniques is hardly supported in a continuous fashion that comprises components and composition as well as the architecture. But whenever it is important to verify or to check the correctness of an implementation with respect to a specification, e.g. in the case of security relevant software, formal techniques equipped with a mathematical semantics have strong advantages compared to less formal techniques.

In [20] D. Garlan lists convincing arguments for the use of formal techniques architecture description languages. In [31] a survey (in German) over the use of formal techniques for the description of software architectures is given. Exemplarily stated there are: Process algebras are used for various architecture description languages, e.g. Darwin [27,28], Wright [2,3] or AEmilia [4], but none of these specifies the component itself. SARA [17] is an early architecture description language using Petri nets for the description of the operational behavior. In [9,10] dualistic Petri nets are proposed. These describe the architecture using abstract representations of parallel process objects. In ZCL [11] is based on Z [36] a set-theoretic specification language. Z schemes are used to describe the architecture structure as well as the dynamic changes. CommUnity [18] and COOL [23] are architecture description languages that are founded on graph transformations. But in COOL no explicit component specification is given.

But there are only a few other approaches with the aim to combine component-based architectures and formal specification techniques in order to have a continuous formal technique. Those examine only a particular specification technique in contrast to our generic approach. E.g. [7] uses the integrated formal specification technique *Korrigan* to specify components and their composition. CommUnity [8,19] is a prototype language for architectural modeling that is founded on graph transformations.

In [13] a transformation-based hierarchical component concept that is generic with respect to the used specification technique and the applied transformation notion, has been presented. This first step to close the gap between formal specification techniques and real life component architectures has been followed by another concept using generic specifications and transformations. In [16,12] connector-based architectures have been introduced to enable the specification of components with multiple provide interfaces that are coordinated by connectors with several require specifications. Both approaches have been successfully instantiated to a variety of specification techniques [14,15]. In this paper a new, even more general approach is introduced which allows the specification of components with several require and provide interfaces, where require interfaces correspond to import interfaces and provide interfaces correspond to export interfaces in the previous transformation-based approaches.

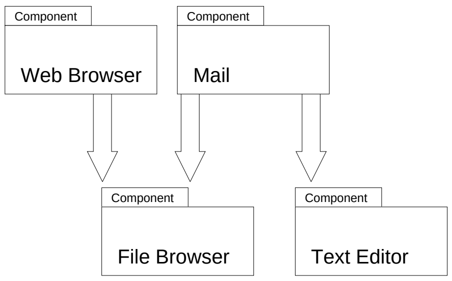

Fig. 1. Sample Architecture Using Multiple Access

Even more important, the approach allows the connection of different requirement specifications of one or more components with the same provide specification. This is a very common scenario in a software developing process. Fig. 1 shows a small architecture containing a web browser, a mail program, a text editor, and a file browser that is accessed by both, the web browser and the mail program. In Sect. 2 it is shown that both accesses operate on the same provide specification and how this is handled within the new approach.

As the above mentioned generic transformation-based approaches, this paper concentrates on a static view of architectures. See Sect. 5 for a discussion of possible extensions handling dynamic architectures. Since the main motivation for a formal approach to component architectures is to enable verification and model checking, it is necessary to calculate the common specification for a given set of component specifications, i.e. the given architecture. This process, explained in detail in Sect. 3, is suitable for many specification techniques and application scenarios.

2 Example: Web Browser Suite

In this section the sample architecture shown in Fig. 1 is explained in detail. We use elementary nets as specification technique. In Sect. 4 we sketch the formal instantiation of our generic framework to this technique.

$$\begin{tikzcd}
COMP\_WB \arrow[dr, Rightarrow] & COMP\_M \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \\
 & COMP\_FB & COMP\_TE
\end{tikzcd}$$

Fig. 2. Architecture Graph of Web Browser Suite

Each component, in general and in the example, consists of a body and a set of provide and require specifications. E.g. the component web browser is given by $COMP\_WB = (REQ\_WB \rightarrow BOD\_WB \Leftarrow PRV\_WB)$ where REQ_WB specifies the require interface, BOD_WB the body and PRV_WB the provide interface. Each of these specifications is given an elementary net. Fig. 2 shows the architecture graph that is the components and their connection.

The more detailed illustration in Fig. 7 shows all specifications of the example's components and all connecting transformations and embeddings. The component index set of the architecture is given by $I = \{WB, M, FB, TE\}$, which is the set of abbreviations for the component names: Web Browser, Mail, File Browser, Text Editor.

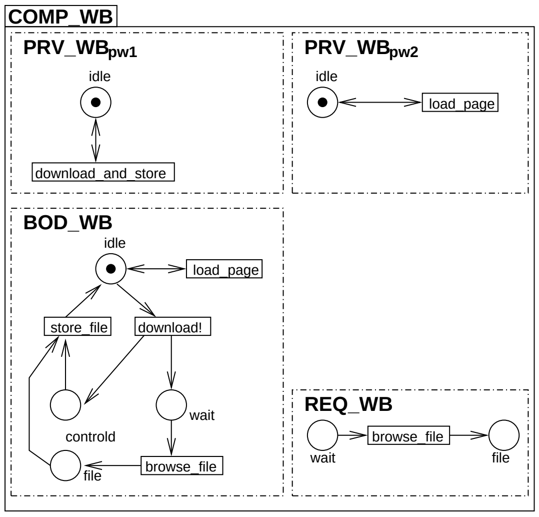

Fig. 3. Web Browser Component

The specification of the web browser component as shown in Fig. 3 contains two provide and a single require interface. The provide interfaces of the web browser component state that this component initially is in a state called idle and two different events can occur: the simple loading of a web page (load_page) or loading and storing a file to disk. Both events lead to the same state, the idle state.

The provide specification nets are refined by the component's body net. The place idle (expressing the initial state of the component) and the load_page transition remain unchanged. The download_and_store transition is replaced by a subnet containing the three places controld, wait and file, and the transitions store_file, download! and browse_file. This subnet models that after each occurrence of the download! event the user has to start a file browser to determine the storage area and the save name of the file. After this selection, the actual download and the file saving are executed. The place controld ensures that the selected file is the result of the started browsing process. This file browsing process and the related places are in the component's only require specification.

The component *COMP_WB* also contains the connections between its provide and require interfaces and the body. For the case of the require interfaces, the corresponding embeddings are quite obvious, and thus omitted. In Fig. 4 the provide interface and the body of component *COMP_WB* are connected. The only place of the provide interfaces is mapped to the same place in the body net. Since the transition load_page remains unchanged in the body, it is

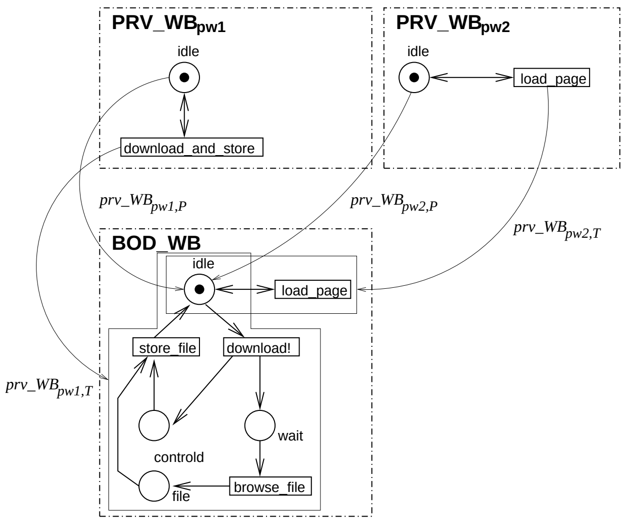

Fig. 4. Transformation *prv_WB*

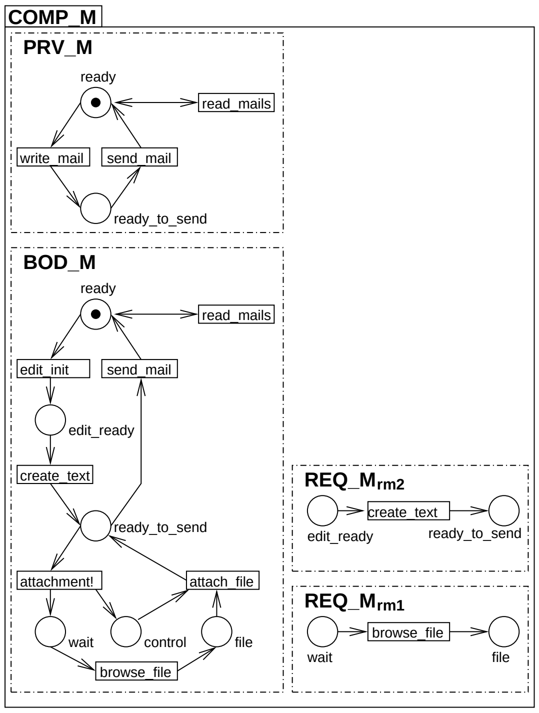

Fig. 5. Mailer Component

mapped to the subnet containing the transition load_page and the only place connected to the transition. The transition download_and_store is mapped to a net containing the whole body except the transition load_page.

The specification of the mailer component is depicted in Fig. 5. Besides a body net it contains a single provide interface and two require interfaces. Initially the provide interface of this component allows two events: read_mails and write_mail. After a mail has been written, the net is enabled to send this message. Both, the send_mail and the read_mails transitions lead to the initial

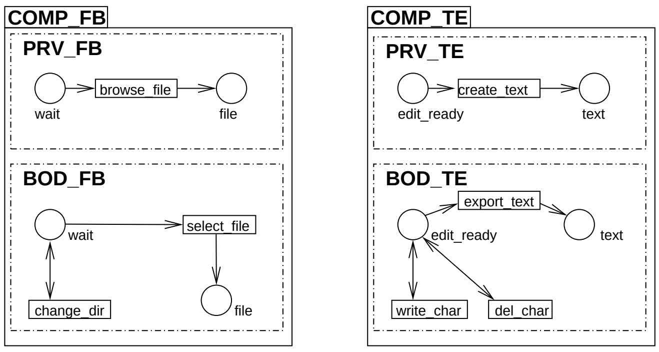

Fig. 6. File Browser and Text Editor Component

state, specified by a marking on place ready.

In the body net of the component the process of writing a mail is refined. First, the mailer starts a text editor which is then used to write the content of the email. Afterwards, the user is enabled to send the mail or to attach a file to it. The latter includes the browsing of a suiting file. Both, the creation of the email content and the browsing of an attachment, are to be provided by the component's environment. This is expressed by the occurrence of the two transitions in the require interfaces.

Fig. 6 shows the components file browser and text editor. Both do not contain a require interface. The simple file browser is specified by only two transitions that offer to change the directory and to select finally the file. The text editor body is specified by three transitions, expressing the possibilities of writing a character, deleting a character and to finally export the written text.

$$\begin{tikzcd}
PRV\_WB \arrow[d, Rightarrow, "prv\_WB"'] & PRV\_M \arrow[d, Rightarrow, "prv\_M"'] & \\
BOD\_WB & BOD\_M & \\
REQ\_WB \arrow[u, "req\_WB"] \arrow[dr, Rightarrow, bend right, "con_{WB,FB}"'] & REQ\_M_{rm1} \arrow[u, "req\_M_{rm1}"] \arrow[d, Rightarrow, "con_{M,rm1,FB}"'] & REQ\_M_{rm2} \arrow[ul, "req\_M_{rm2}"'] \arrow[d, Rightarrow, "con_{M,rm2,TE}"'] \\
 & PRV\_FB \arrow[d, Rightarrow, "prv\_FB"'] & PRV\_TE \arrow[d, Rightarrow, "prv\_TE"'] \\
 & BOD\_FB & BOD\_TE
\end{tikzcd}$$

Fig. 7. Detailed Architecture of the Web Browser Suite

Fig. 7 illustrates the architecture diagram of the web browser suite. It involves all specifications, transformations, and embeddings, but it disregards all the specific elementary nets given in Figures 3-6. The abstraction of this diagram is the architecture graph in Fig. 2.

In this sample architecture all but one connecting transformation are identity transformations, i.e. we have equality of the corresponding require and provide interfaces. The transformation $con_{M,rm2,TE}$ shown in Fig. 8 is different, since it actually applies the possibility to rename places along transformations of marked elementary nets. In general, our framework offers the possibility to connect require and provide interfaces by refining transformations. For the case of elementary nets this includes the possibility to replace transitions by subnets.

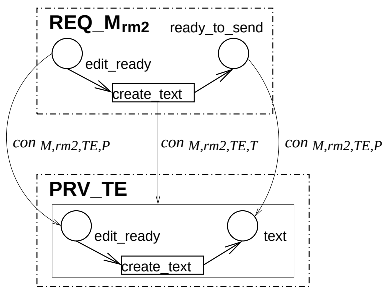

Fig. 8. Transformation $con_{M,rm2,TE}$

Since the main motivation for applying formal techniques to software engineering is verification we need to construct from the given components a single specification that can be verified with the corresponding tools of that specification technique. In the next section we define how components can be composed to larger ones. In the case of our sample architecture, the repeated application of the composition operation yields a component that contains the whole behavior of the browser suite. Fig. 9 shows the body of the resulting component.

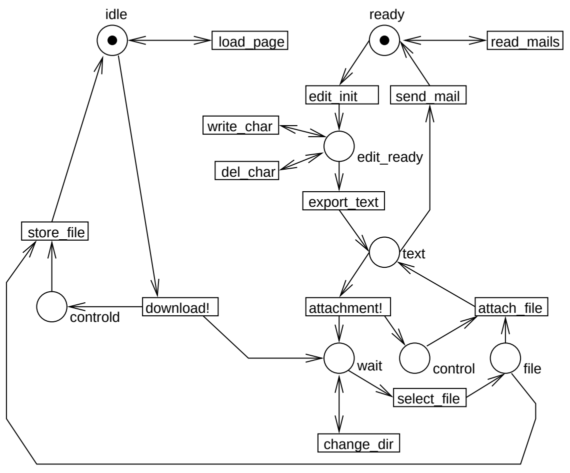

Fig. 9. Body of Composed Browser Suite

After having derived the complete specification, we now can start verification or model checking, respectively, with respect to a given requirement specification, but this is not within the scope of this paper.

3 The Generic Framework

One central aim of this work is to define generic notions of components and composition operations capable of handling multiple access scenarios, as shown in the example of the previous section. As the approaches [13] and [16,12] this work applies generic specifications, embeddings and transformations to form components. Since not all classes of embeddings and transformations are suitable for this purpose we have to state some general requirements first. The validity of these requirements needs to be proven in the concrete specification technique when instantiating the generic concept.

3.1 General Requirements

Our generic technique requires a defined class of specifications, corresponding transformations and embeddings. Since the transformations are used in the framework to establish the connection between provide interfaces and the actual component specification, the component body, it is sensible to assume that the transformations define a class of refinements for the specifications. Since there exist so many notions of refinement, even for single specification techniques, this assumption should not be further formalized at the abstract level - but it has to be clarified when the concept is instantiated. In Sect. 2 we applied a refinement notion for elementary nets that allows mapping single transitions to whole subnets (see Sect. 4 for details).

For both, the transformations and the embeddings, we require a composition operation and a special identity instance. Moreover, it is necessary that the class of embeddings defines a subclass of the transformations, i.e. we require a mapping $trafo : EMB \to TRAFO$ that selects a transformation for each embedding.

The extension property defined below is well-known from [12] and [13]. It states that a single transition can be applied to a larger context.

Definition 3.1 Extension Property Given an embedding $e : SPEC\_R \to SPEC$ and a transformation $t : SPEC\_R \Longrightarrow SPEC'$. Now there is a selected transformation $t' : SPEC \Longrightarrow SPEC'$ and a selected embedding $e' : SPEC\_R' \to SPEC'$, such that diagram (1) in Fig. 10 becomes an extension diagram. In case of t also being an embedding, we require the existence of a unique extension diagram (2), called mutual extension diagram.

$$\begin{tikzcd}
SPEC\_R \arrow[r, "e"] \arrow[d, Rightarrow, "t"'] \arrow[dr, phantom, "(1)"] & SPEC \arrow[d, Rightarrow, "t'"] & & SPEC\_R \arrow[r, "e"] \arrow[d, "t"'] \arrow[dr, phantom, "(2)"] & SPEC \arrow[d, "t'"] \\
SPEC\_R' \arrow[r, "e'"'] & SPEC' & & SPEC\_R' \arrow[r, "e'"'] & SPEC'
\end{tikzcd}$$

Fig. 10. Extension

The multiple extension defined below expresses the possibility to apply a set of transformations to a larger context within a single transformation. It differs from the parallel extension used in [16] and [12] by allowing given transformations with the same codomain only, and it contains the extension defined above as a special case. In general, this construction is not available for all families of embeddings and corresponding transformations. Intuitively, such families allow multiple extension, if the boundary of all embeddings is preserved i.e. the transformations do not delete or rewrite parts that are needed to maintain a well-formed specification, and all overlappings with respect to the embeddings are transformed uniquely.

Definition 3.2 Multiple Extension Given an index set I, a corresponding family of embeddings $e = (e_i : \mathit{SPEC\_R}_i \to \mathit{SPEC})_{i \in I}$ and a family of transformations $tr = (tr_i : \mathit{SPEC\_R}_i \Longrightarrow \mathit{SPEC\_R})_{i \in I}$. Now e and tr allow multiple extension, if there exist a selected transformation $t : \mathit{SPEC} \Longrightarrow \mathit{SPEC}'$ and a single embedding $e' : \mathit{SPEC\_R} \to \mathit{SPEC}'$. We call diagram $(1_i)_{i \in I}$ multiple extension diagram.

$$\begin{tikzcd}
\mathit{SPEC\_R}_i \arrow[r, "e_i"] \arrow[d, Rightarrow, "tr_i"'] \arrow[dr, phantom, "(1_i)_{i\in I}"] & \mathit{SPEC} \arrow[d, Rightarrow, "t"] \\
\mathit{SPEC\_R} \arrow[r, "e'"'] & \mathit{SPEC}'
\end{tikzcd}$$

Fig. 11. Multiple Extension

Definition 3.3 Compatibility of Embeddings with Multiple Extension A family of embeddings $(e_i : \mathit{SPEC\_R}_i \to \mathit{SPEC})_{i \in I}$ is compatible with multiple extension, if for each multiple extension diagram (1) with a family of transformations $(tr_i : \mathit{SPEC\_R}_i \Longrightarrow \mathit{SPEC\_R})_{i \in C \subseteq I}$, we have for the family of embeddings $(e_j : \mathit{SPEC\_R}_j \to \mathit{SPEC})_{j \in I \setminus C}$ a selected family of embeddings $(e'_j : \mathit{SPEC\_R}_j \to \mathit{SPEC}')_{j \in I \setminus C}$.

$$\begin{tikzcd}
\mathit{SPEC\_R}_i \arrow[r, "e_i"] \arrow[d, Rightarrow, "tr_i"'] \arrow[dr, phantom, "(1)"] & \mathit{SPEC} \arrow[d, Rightarrow, "t"] \arrow[r, phantom, "(2)" below] & \mathit{SPEC\_R}_j \arrow[l, "e_j"'] \arrow[dl, "e'_j"] \\
\mathit{SPEC\_R} \arrow[r, "e'"'] & \mathit{SPEC}' &
\end{tikzcd}$$

Fig. 12. Compatibility of Embeddings

We require the existence of a subclass D of all families of embeddings such that all elements in this subclass are compatible with multiple extension. Moreover, we require that D is closed under multiple extensions. I.e. if $(e_i : \mathit{SPEC\_R}_i \to \mathit{SPEC})_{i \in I}$ in Fig. 12 is in class D and (1) is an extension diagram and we have embeddings e'_j then also $(e'_j : \mathit{SPEC\_R}_j \to \mathit{SPEC}')_{j \in I \setminus C} \cup \{e' : \mathit{SPEC\_R} \to \mathit{SPEC}'\}$ is in class D. In the corresponding instantiation this can be achieved by defining D by non-overlapping embeddings. Note that the instantiation has to define which transformations and embeddings make up a multiple extension diagram.

Moreover, we assume horizontal and vertical composition of multiple extension diagrams: Given diagrams (1), (2), (3) in Fig. 13 with $i \in I$ and $j \in J$. Now (1+3) and (2+3) have to be multiple extension diagrams if (1) and (2) are multiple extension diagrams and (3) is an extension diagram.

$$\begin{tikzcd}
 & SPEC\_R_i \arrow[r, "er_i"] \arrow[d, Rightarrow, "tr_i"'] \arrow[dr, phantom, "(1)"] & SPEC \arrow[d, Rightarrow, "t"'] \\
SPEC\_C_j \arrow[r, "ec_j"] \arrow[d, Rightarrow, "tr_j"'] \arrow[dr, phantom, "(2)"] & SPEC\_R \arrow[r, "er'"] \arrow[d, Rightarrow, "t'"'] \arrow[dr, phantom, "(3)"] & SPEC' \arrow[d, Rightarrow, "t''"'] \\
SPEC\_C \arrow[r, "ec'_j"] & SPEC\_R' \arrow[r, "er''"] & SPEC''
\end{tikzcd}$$

Fig. 13. Composition of Extension Diagrams

3.2 Components and Composition

Based on the requirements explained above, we are now able to define component specifications and the corresponding composition operation.

Definition 3.4 Component A component specification $COMP = (BOD, REQ, PRV, req, prv)$ consists of a body specification BOD, a family of require specifications $REQ = (REQ_i)_{i \in I}$ for some index set I, a family of provide specifications $PRV = (PRV_j)_{j \in J}$ for some index set J and of suiting families of embeddings $req = (req_i : REQ_i \rightarrow BOD)_{i \in I}$ and transformations $prv = (prv_j : PRV \Longrightarrow BOD)_{j \in J}$, respectively, where we require that the family of embeddings is in class D and thus compatible with multiple extension in the sense of Def. 3.3.

Note that the components in our example in Sect. 2 fit into this abstract definition. The web browser component in Fig. 3 contains two provide specifications, a body specification and a single require specification. The corresponding transformations of the provide interfaces are shown in Fig. 4.

Next, we summarize the conditions ensuring that a given set of connected components can be reduced to a single component. According to [16,12] we call such a set an *architecture*. An architecture A is a set of components $COMPS(A) = (COMP\_i)_{i \in I}$ and corresponding connecting transformations $CONS(A)$ that fulfill the properties listed below. Each architecture A can be illustrated by an *architecture graph* G_A (e.g. as for the web browser suite in Fig. 2), obtained by shrinking A to a graph representation that contains nodes labeled by the component names and edges labeled by the connecting transformations.

- There are no isolated components in the architecture.

- Each requirement specification is the source of at most one connecting transformation.
- For each component we require that its embedding of the require interfaces and the connected realizing transformations allow multiple extension. (i.e. $(\mathit{prv\_2}_y \circ \mathit{con}_{1,j,2,y})_{j\in I\_1, y\in J\_C2, \mathit{con}_{1,j,2}\in CONS(A)}$ allow multiple extension)
- There are no cycles in the graph obtained by representing single specifications by nodes and transformations and embeddings by non-directed edges.

In [16,12] it has been shown that architectures can be reduced to a single component, if the applied composition operations yield unique results independing of their application order. This is the case for the operations defined below.

The hierarchical composition with multiple interfaces defined below connects a single providing component to a single requiring component, possibly via different provide and require interfaces. Intuitively, the requiring component is glued with the providing component over the provide interfaces accessed by the requiring component.

Definition 3.5 Hierarchical Composition with Multiple Interfaces Given a requiring component $COMP\_R = (BOD\_R, REQ\_R, PRV\_R, \mathit{req\_R}, \mathit{prv\_R})$ and a providing component $COMP\_P = (BOD\_P, REQ\_P, PRV\_P, \mathit{req\_P}, \mathit{prv\_P})$ with index sets I_R, J_R and I_P, J_P for the require and provide interfaces of the requiring component and the providing component, respectively. We denote the index set of the require interfaces actually connected

$$\begin{tikzcd}
 & & PRV\_R_j \arrow[d, Rightarrow, "\mathit{prv\_R}_j"] & \\
 & REQ\_R_i \arrow[r, "\mathit{req\_R}_i"] \arrow[d, Rightarrow, "\mathit{con}_i"] & BOD\_R \arrow[dd, Rightarrow, "\mathit{xcon}'"] & REQ\_R_z \arrow[l, "\mathit{req\_R}_z"'] \arrow[ddl, "\mathit{req\_R}'_z"] \\
PRV\_P_x \arrow[dr, Rightarrow, "\mathit{prv\_P}_x"] & PRV\_P_{l(i)} \arrow[d, Rightarrow, "\mathit{prv\_P}_{l(i)}"] \arrow[r, phantom, "(1_i)"] & & \\
REQ\_P_y \arrow[r, "\mathit{req\_P}_y"] & BOD\_P \arrow[r, "\mathit{req\_R}'"] & BOD\_G &
\end{tikzcd}$$

Fig. 14. Hierarchical Composition with Multiple Requirements

with the providing component by $C\_R \subseteq I\_R$. Given corresponding connecting transformations $\mathit{con} = \mathit{con}_i : REQ\_R_i \Longrightarrow PRV\_P_{l(i)}$ with $i \in C\_R \subseteq I\_R$ and $l(i) \in J\_P$, such that the family of embeddings $\mathit{req\_R}$ and the family of composed transformations $\mathit{xcon}_i = (\mathit{prv\_P}_{l(i)} \circ \mathit{con}_i)_{i\in C\_R}$ allow multiple extension. The mapping $l : I\_R \rightarrow J\_P$ has to be injective. The index sets of all components are disjoint. In the first step we can derive a multiple extension diagram $(\mathit{1}_i)_{i\in C\_R}$ with selected transformation xcon' and embedding $\mathit{req\_R}'$. The compatibility of the embeddings $(\mathit{req\_R}_i)_{i\in I}$ with respect to multiple extension, which is given by the component definition, yields a set of embeddings

$(req\_R'_z)_{z\in I\_R\setminus C\_R}$, such that $(req\_R'_z)_{z\in I\_R\setminus C\_R} \cup \{req\_R'\}$ is again in class D and thus compatible with respect to multiple extension. Now we define the result of the *Hierarchical Composition with Multiple Interfaces* (short: composition) by

$$COMP\_R \circ_{con} COMP\_P =$$
$$COMP\_G = (BOD\_G, REQ\_G, PRV\_G, req\_G, prv\_G),$$

where the index sets of the requirements and provisions of the new component are defined as $I\_G = (I\_R \setminus C\_R) \cup I\_P$ and $J\_G = J\_R \cup J\_P$, respectively. And we have:

$$\begin{aligned}
REQ\_G &= (REQ\_R_z)_{z\in I\_R\setminus C\_R} \cup REQ\_P\\
req\_G &= (req\_R'_z)_{z\in I\_R\setminus C\_R} \cup (req\_R' \circ req\_P_y)_{y\in I\_R}\\
PRV\_G &= PRV\_R \cup PRV\_P\\
prv\_G &= (xcon' \circ prv\_P_j)_{j\in J\_R} \cup (trafo(req\_R') \circ prv\_P_x)_{x\in J\_P}
\end{aligned}$$

Note that the family of embeddings $(req\_G_i)_{i\in I\_G}$ is again in class D. In Fig. 15 the elements of the resulting component are depicted in detail.

$$\begin{tikzcd}
 & & PRV\_R_j \arrow[dd, Rightarrow, "xcon'\circ prv\_R_j"] & \\
PRV\_P_x \arrow[dr, Rightarrow, "prv\_P_x"] & PRV\_P_{l(i)} \arrow[d, Rightarrow, "prv\_P_{l(i)}"] & & REQ\_R_z \arrow[dl, "req\_R'_z"] \\
REQ\_P_y \arrow[r, "req\_P_y"] & BOD\_P \arrow[r, "req\_R'"] \arrow[r, Rightarrow, shift right=1.5, "trafo(req\_R')"'] & BOD\_G &
\end{tikzcd}$$

Fig. 15. Result of Composition

Since we allow different components accessing the same provide interface, all provide interfaces are preserved by the composition.

Fig. 16 shows the composition of the mailer and the text editor component of our example from Sect. 2. Note that only the transformations prv_TE, $con_{M,rm2,TE}$ and the embedding req_M_{rm2} are shown in detail.

The hierarchical composition with multiple interfaces is independent of its application order. Since there are three possibilities of overlappings for two composition steps, we present three different theorems: associativity of composition, compatibility of composition I and II.

Whenever we have two connections crossing the same level of a given component architecture, we also offer a parallel composition, which constructs the result of two compositions within a single step. This is the case in the Theorems 3.8 and 3.9.

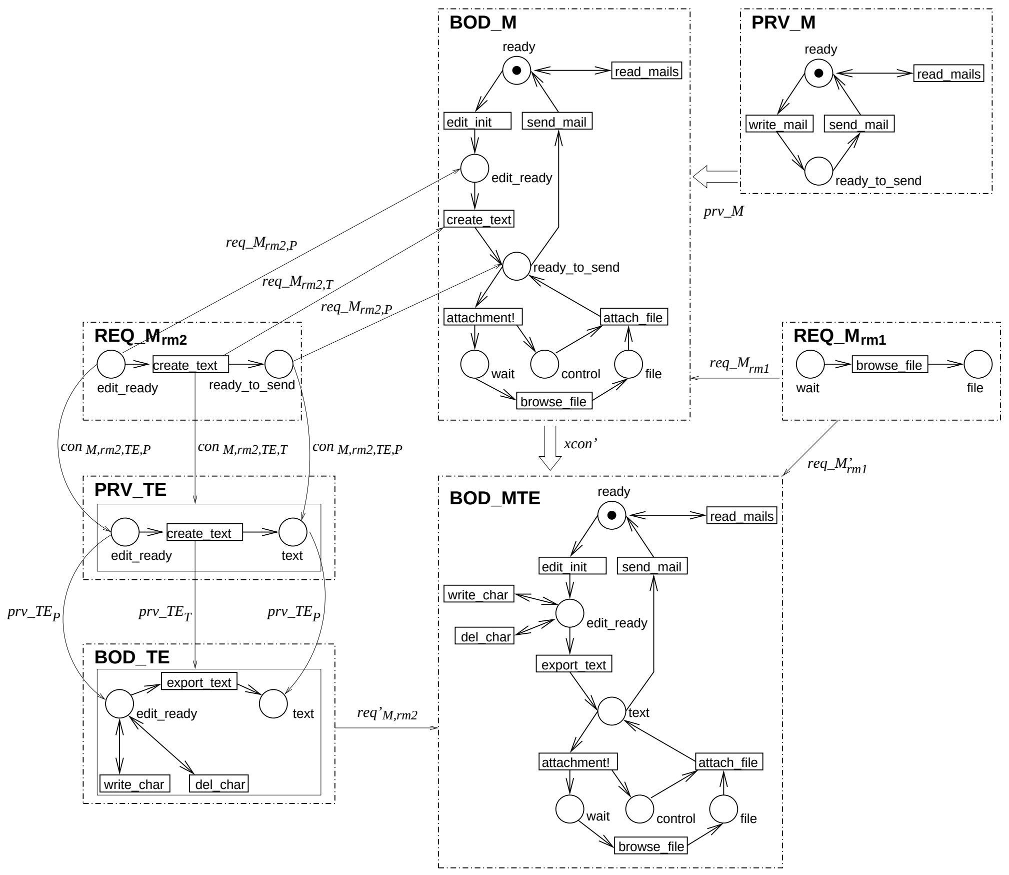

Fig. 16. Composition of Mailer and Text Editor

$$\begin{tikzcd}
COMP\_1 \arrow[d, Rightarrow, "con\_1"] \\
COMP\_2 \arrow[d, Rightarrow, "con\_2"] \\
COMP\_3
\end{tikzcd}$$

Fig. 17. Architecture Graph G_{A1}

Theorem 3.6 ***Associativity of Composition*** *Given three components $COMP\_i = (BOD\_i, REQ\_i, PRV\_i, req\_i, prv\_i)$ for $i \in \{1, 2, 3\}$ as shown in the architecture graph G_{A1} and families of transformations $con\_1 = (con\_1_i : REQ\_1_i \Longrightarrow PRV\_2_{l(i)})_{i \in C\_1}$, $con\_2 = (con\_2_k : REQ\_2_k \Rightarrow PRV\_3_{l(k)})_{k \in C\_2}$, for some $C\_1 \subseteq I\_1, C\_2 \subseteq I\_2$, where I_1 and I_2 denote the index sets of the require interfaces of COMP_1 and COMP_2, such that the pairs of families $(prv\_2_{l(i)} \circ con\_1_i)_{i \in C\_1}$, req_1 and $(prv\_3_{l(k)} \circ con\_2_k)_{k \in C\_2}$, req_2 allow* *multiple extension, each. Then we have the following associativity law:*

$$(COMP\_1 \circ_{con\_1} COMP\_2) \circ_{con\_2} COMP\_3 =$$
$$COMP\_1 \circ_{con\_1} (COMP\_2 \circ_{con\_2} COMP\_3)$$

Proof. Fig. 18 shows the given components and connecting transformations in detail, where we have:

$$\begin{array}{ll} j \in J\_1, & l\_1 : C\_1 \to J\_2 \text{ injective,} \\ i \in C\_1 \subseteq I\_1, & k \in C\_2 \subseteq I\_2, \\ i' \in I \setminus C\_1, & k' \in I\_2 \setminus C\_2, \text{ and} \\ j' \in J\_2 \setminus l(C\_1), & m \in I\_3, x \in J\_3. \end{array}$$

$$\begin{tikzcd}
 & & & PRV\_1_{j} \arrow[d, Rightarrow, "prv\_1_{j}"] & \\
 & & REQ\_1_{i} \arrow[r, "req\_1_{i}"] \arrow[d, Rightarrow, "con\_1_{i}"] & BOD\_1 \arrow[dd, Rightarrow, "xcon"] & REQ\_1_{i'} \arrow[l, "req\_1_{i'}"'] \arrow[dl, bend left, "req\_1'_{i'}"] \\
REQ\_2_{k'} \arrow[drr] & PRV\_2_{j'} \arrow[dr, Rightarrow, "prv\_2_{j'}"] & PRV\_2_{l\_1(i)} \arrow[d, Rightarrow, "prv\_2_{l(i)}"] \arrow[r, phantom, "(1)"] & & \\
 & REQ\_2_{k} \arrow[r, "req\_2_{k}"] \arrow[d, Rightarrow, "con\_2_{k}"] & BOD\_2 \arrow[r] \arrow[dd, Rightarrow] & BOD\_12 \arrow[dd, Rightarrow] & \\
PRV\_3_{x} \arrow[dr, Rightarrow] & PRV\_3_{l\_2(k)} \arrow[d, Rightarrow] \arrow[r, phantom, "(2)"] & {} \arrow[r, phantom, "(3)"] & {} & \\
REQ\_3_{m} \arrow[r, "req\_3_{m}"'] & BOD\_3 \arrow[r] & BOD\_23 \arrow[r] & BOD\_123 &
\end{tikzcd}$$

Fig. 18. Associativity of Composition

We are able to construct the multiple extension diagrams (1) and (2) due to the assumptions. The extension diagram (3) exists, because there are no properties required for this construction. The body of the left side of our equation, $(COMP_1 \circ_{con\_1} COMP\_2) \circ_{con\_2} COMP\_3$, is constructed by the following steps: First, we have to construct extension diagram (1) to resolve *con_1*. We know that (2) and (3) are multiple extension diagrams, thus we can construct (2+3) and resolve *con_2*. For the construction of the body of the left side of our equation, $COMP\_1 \circ_{con\_1} (COMP\_2 \circ_{con\_2} COMP\_3)$, the first step is to resolve *con_2* using multiple extension diagram (2), and afterwards resolving con_1 by multiple extension diagram (1+3). Since extension yields unique resulting specifications and transformations, we obtain a unique body BOD_123. □

3.3 Compatibility of Composition

In order to ensure a unique reduction of architectures we need to prove that for all kinds of overlappings of components the result of several composition steps is independent of the order of the composition steps. In Thm. 3.6 this was shown for overlappings along an hierarchy. This section deals with composition steps that include overlappings of components of the same hierarchical level. Fig. 19 shows such an architecture. For this case, the result of the given compositions can be constructed within one parallel step. We can prove that the result of this *parallel composition* is equal to the sequential composition independent of the ordering.

$$\begin{tikzcd}
COMP\_1 \arrow[dr, Rightarrow, "con\_1"] & & COMP\_2 \arrow[dl, Rightarrow, "con\_2"'] \\
& COMP\_3 &
\end{tikzcd}$$

Fig. 19. Architecture Graph G_{A2}

Definition 3.7 Parallel Composition Given three components $COMP\_i = (BOD\_i, REQ\_i, PRV\_i, req\_i, prv\_i)$ for $i \in \{1, 2, 3\}$ and families of transformations $con\_1 = (con\_1_i : REQ\_1_i \Longrightarrow PRV\_3_{l(i)})_{i\in C\_1}$, $con\_2 = (con\_2_k : REQ\_2_k \Longrightarrow PRV\_3_{l(k)})_{k\in C\_2}$, for some $C\_1 \subseteq I\_1, C\_2 \subseteq I\_2$, such that the pairs $(prv\_3_{l(i)} \circ con\_1_i)_{i\in C\_1}$, req_1 and $(prv\_3_{l(k)} \circ con\_2_k)_{k\in C\_2}$, req_2 allow multiple extension, each. Then we construct the multiple extension diagrams (1) and (2) in Fig. 20. Diagram (3) is constructed as mutual extension diagram, including the resulting body BOD_123. The result of the parallel composition is given by

$$(COMP\_1, COMP\_2) \circ_{(con\_1,con\_2)} COMP\_3 =$$
$$COMP\_123 = (REQ\_123, PRV\_123, req\_123, prv\_123),$$

where

$$\begin{aligned}
REQ\_123 &= (REQ\_1_{i'})_{i'\in I\_1\setminus C\_1} \cup (REQ\_2_{k'})_{k'\in I\_2\setminus C\_2},\\
req\_123 &= (req\_2'' \circ req\_1'_{i'})_{i'\in I\_1\setminus C\_1} \cup (req\_1'' \circ req\_2'_{k'})_{k'\in I\_2\setminus C\_2},\\
PRV\_123 &= (PRV\_1_j)_{j\in J\_1} \cup (PRV\_2_{j'})_{j'\in J\_2} \cup (PRV\_3_x)_{x\in J\_3},\\
prv\_123 &= (trafo(req\_2'') \circ xcon\_1' \circ prv\_1_j)_{j\in J\_1} \cup\\
&\quad (trafo(req\_1'') \circ xcon\_2' \circ prv\_2_{j'})_{j'\in J\_2} \cup (prv\_3_x)_{x\in J\_3}.
\end{aligned}$$

$$\begin{tikzcd}
 & PRV\_1_j \arrow[d, Rightarrow, "prv\_1_j"] & & & & & & PRV\_2_{j'} \arrow[d, Rightarrow, "prv\_2'_j"] & \\
REQ\_1_{i'} \arrow[r] \arrow[dd, bend right, "req\_1'_{i'}"] & BOD\_1 \arrow[dd, Rightarrow] & REQ\_1_i \arrow[l] \arrow[d, Rightarrow, "con\_1_i"] \arrow[l, phantom, "(1)", shift right=20] & & & & REQ\_2_k \arrow[r] \arrow[d, Rightarrow, "con\_2_k"'] & BOD\_2 \arrow[dd, Rightarrow] \arrow[l, phantom, "(2)", shift left=20] & REQ\_2_{k'} \arrow[l] \arrow[dd, bend left, "req\_2'_{k'}"'] \\
 & & & PRV\_3_{l_1(i)} \arrow[dr, Rightarrow, "prv\_3_{l_1(i)}"'] & PRV\_3_x \arrow[d, Rightarrow, "prv\_3_x"] & PRV\_3_{l_2(k)} \arrow[dl, Rightarrow, "prv\_3_{l_2(k)}"] & & & \\
 & BOD\_13 \arrow[drrr, "req\_2''"'] & & & BOD\_3 \arrow[lll, "req\_1'"] \arrow[rrr, "req\_2'"'] \arrow[d, phantom, "(3)"] & & & BOD\_23 \arrow[dlll, "req\_1''"] & \\
 & & & & BOD\_123 & & & &
\end{tikzcd}$$

Fig. 20. Parallel Composition

Theorem 3.8 ***Compatibility of Composition I*** *Given the same components and connecting transformations as in the definition above, then we have the following compatibility law:*

$$COMP\_1 \circ_{con\_1} (COMP\_2 \circ_{con\_2} COMP\_3) =$$
$$COMP\_2 \circ_{con\_2} (COMP\_1 \circ_{con\_1} COMP\_3) =$$
$$(COMP\_1, COMP\_2) \circ_{(con\_1, con\_2)} COMP\_3$$

The web browser suite presented in Sect. 2 involves such a situation: The web browser component and the mailer component access the file browser component, as shown in Figures 2 and 7.

Proof. Fig. 20 shows the given setting in detail, where the mappings $l_1 : I\_1 \to J\_3$ and $l_2 : I\_2 \to J\_3$ are injective each, but their codomains are not assumed to be disjoint. First, we construct the result of the parallel composition that resolves both connection in a single step. In this case, we start by computing the multiple extension diagrams (1) and (2), which exist due to the assumption of multiple extension for $prv\_3_{l_1(i)} \circ con\_1_i$ and req_1 as well as for $prv\_3_{l_2(k)} \circ con\_1_k$ and req_2. Diagram (3) is constructed as mutual extension diagram including the resulting body BOD_123. In case of processing only the composition along con_2 in the first place, we obtain the extension diagram (2). Afterwards we construct the multiple extension diagram (1') as depicted in Fig. 21. This diagram is constructed from the same given transformations and embeddings as diagram (1+3) in the case of the parallel composition explained above. This implies $BOD\_231 = BOD\_123$.

Analogously we construct BOD_13 in the multiple extension diagram (1). Then we construct the extension diagram (2') which is equal to (2+3) in the parallel composition. This implies $BOD\_132 = BOD\_123 = BOD\_231$. Ad-

$$\begin{tikzcd}
REQ\_1_i \arrow[r, "req\_1_i"] \arrow[d, Rightarrow, "con\_1_i"'] & BOD\_1 \arrow[dd, Rightarrow] & & REQ\_2_k \arrow[r, "req\_2_k"] \arrow[d, Rightarrow, "con\_2_k"'] & BOD\_2 \arrow[dd, Rightarrow] \\
PRV\_3_{l_1(i)} \arrow[r, phantom, "(1')"] \arrow[d, Rightarrow, "trafo(req\_1')\circ prv\_3_{l_1(i)}"'] & & & PRV\_3_{l_2(k)} \arrow[r, phantom, "(2')"] \arrow[d, Rightarrow, "trafo(req\_2')\circ prv\_3_{l_2(k)}"'] & \\
BOD\_23 \arrow[r, "req\_1''"] & BOD\_231 & & BOD\_13 \arrow[r, "req\_2''"] & BOD\_132
\end{tikzcd}$$

Fig. 21. Stepwise Composition

ditionally, in all three cases we obtain the same families of provide and require interfaces and the corresponding connections, because the disjoint index sets of the given components imply independence of the composition order. □

$$\begin{tikzcd}
& COMP\_1 \arrow[dl, Rightarrow, "con\_1"'] \arrow[dr, Rightarrow, "con\_2"] & \\
COMP\_2 & & COMP\_3
\end{tikzcd}$$

Fig. 22. Architecture Graph G_{A3}

Theorem 3.9 ***Compatibility of Composition II*** *Given three components $COMP\_i = (BOD\_i, REQ\_i, PRV\_i, req\_i, prv\_i)$ for $i \in \{1,2,3\}$ and families of transformations $con\_1 = (con\_1_i : REQ\_1_i \Longrightarrow PRV\_2_{l(i)})_{i\in C\_1}$, $con\_2 = (con\_2_k : REQ\_1_k \Longrightarrow PRV\_3_{l(k)})_{k\in C\_2}$, for some $C\_1, C_2 \subseteq I\_1$ such that the pairs $(prv\_2_{l(i)} \circ con\_1_i)_{i\in C\_1}$, req_1 and $(prv\_3_{l(k)} \circ con\_2_k)_{k\in C\_2}$, req_1 allow multiple extension, each. Then we have the following compatibility law:*

$$(COMP_1 \circ_{con\_1} COMP\_2) \circ_{con\_2} COMP\_3 =$$

$$(COMP\_1 \circ_{con\_1} COMP\_3) \circ_{con\_2} COMP\_2) =$$

$$COMP\_1 \circ_{(con\_1, con\_2)} (COMP\_2, COMP\_3)$$

In our example in Sect. 2 this situation occurs as well. Figures 2 and 7 show that the mailer component accesses both, the file browser component and the text editor component.

A full proof of the theorem is given in [26]. Here, we only sketch its main idea. In [16] the *parallel extension* diagram was introduced that embeds independent transformations into a common larger context. Moreover, it offers a special case with all but one given transformation being identities. This is also the case here, because the require interfaces of *COMP_1* are disjoint. Composing those diagrams yields the intended uniqueness of the body construction.

4 Instantiation to Elementary Nets

In this section we show that the specification technique of elementary nets [35] fits into our generic framework. This includes the definition of embeddings and transformations, and based on that, the construction of the multiple extensions.

The hierarchic transformation-based concept in [13] and the connector component framework in [16] have been instantiated with a variety of specification techniques: HLR-systems and algebraic specifications in [25], Petri nets in [15] and UML diagrams in [12]. Since our approach is a generalization of those two concepts the instantiations can be easily adopted to the new concept.

Elementary transition systems are a special notion of Petri nets, allowing only arcs and place weights of arity one. We use the algebraic notion of Petri nets as given in [29] and extend it by the initial marking. This enables us to use a set based representation of the pre and post functions of the transitions of the nets.

An elementary net $N = (P, T, pre, post, m)$ consists of a set of places P and a set of transitions T. The functions $pre, post\colon T \to \mathcal{P}(P)$ represent the connecting arcs, and the set $m \subseteq P$ contains all initially marked places. Plain morphisms$f : N_1 \to N_2$ between elementary nets are mappings of places $f_P : P_1 \to P_2$ and of transitions $f_T : T_1 \to T_2$ that are compatible with the pre and $post$, i.e. $f_P \circ pre_1(t) = pre_2 \circ f_T(t)$ and analogously for $post$. The mapping has to preserve and reflect the initial marking, i.e. $f_P(m_1) \subseteq m_2$ and $m_2 \setminus f_P(m_1) \subseteq P_2 \setminus f_P(P_1)$. This category EN_{plain} has pushouts.

Embeddings are injective morphisms. The following notion of transformation of elementary nets is an adaption of the substitution morphisms of place/transition nets in [30]. These morphisms replace transitions of the original net by whole subnets in the target net and map places injectively. Again, the markings are preserved and reflected. More precisely, a substitution morphism $s = (s_P, s_T) : N_1 \to N_2$ with $N_i = (P_i, T_i, pre_i, post_i, m_i)$ for $(i = 1, 2)$ is given by an injective mapping of places $s_P : P_1 \to P_2$ and a mapping $s_T : T_1 \to \mathcal{P}(N_2)$ with $s_T(t) := N_2^t = (P_2^t, T_2^t, pre_2, post_2, m_2) \subseteq N_2$ where $pre_2, post_2$ and m_2 are restricted to the subset of transitions T_2^t and the subset of places P_2^t. Again we have preservation and reflection of the marked places, i.e. $s_P(m_1) \subseteq m_2$ and $m_2 \setminus s_P(m_1) \subseteq P_2 \setminus s_P(P_1)$. Composition is well-defined analogously to [30]. So, we have the category EN. Similar to [30] plain morphisms are a special case of substitution morphisms.

We have the extension properties as required in Def. 3.1, because there are in the category EN pushouts of embeddings with substitution morphisms as well as pushouts of plain morphisms only. The abstract framework requires a class D with compatibility of embeddings with multiple extension (see Def 3.3). For this instantiation with elementary nets a family of embeddings $e_i : N_i \to N$ has no overlappings, if the codomain of the embeddings

$e_i(N_i)$ are pairwise disjoint.

Then we have multiple extension as required in Def. 3.2. Basically we glue N_R with N together by replacing the embeddings of N_R_i by their substitution subnets $tr_i(N\_R_i)$ in N_R. In the detailed proof [26] we have given the construction in categorical terms, based on the following diagram in EN:

$$\begin{tikzcd}
N\_R_i \arrow[r, "e_i"] \arrow[d, Rightarrow, "tr_i"'] \arrow[dr, phantom, "(\mathbf{i})"] & N \arrow[d, Rightarrow, "\widehat{tr_i}"] & & \\
N\_R \arrow[r, "e'_i"'] & \widehat{N_i} \arrow[r, "\widehat{e_i}"'] & \widetilde{N} \arrow[r, "\widetilde{e}"'] & N'
\end{tikzcd}$$

First, we construct i pushout diagrams $(\mathbf{i})$. Next we construct the star-pushout $N\_R_i \xrightarrow{e'_i} \widehat{N_i} \xrightarrow{\widehat{e_i}} \widetilde{N}$ and subsequently the star-coequalizer $N \overset{\widehat{e_i}\circ\widehat{tr_i}}{\Longrightarrow} \widetilde{N} \xrightarrow{\widetilde{e}} N'$. Then we have the unique $e' = \widetilde{e}\circ\widehat{e_i}\circ e'_i$ and the unique $tr = \widetilde{e}\circ\widehat{e_i}\circ\widehat{tr_i}$ We show in [26] that this star-coequalizer exists and that e' is a well-defined embedding.

Compatibility of embeddings with multiple extension as required in Def. 3.3 we have for families of embeddings $(e_i : N_i \to N)_{i\in I}$ that have no overlappings, because the pushout and coequalizer constructions leave those parts that are not in the codomain $(e_i(N\_R_i))_{i\in C\subseteq I}$ unchanged, especially the codomain of $(e_j(N\_R_j))_{j\in I\setminus C}$. Hence there is the family of embeddings $(e'_j : N\_R_j \to N')_{j\in I\setminus C}$ that remains non-overlapping, see [26].

$$\begin{tikzcd}
N\_R_i \arrow[r, "e_i"] \arrow[d, Rightarrow, "tr_i"'] \arrow[dr, phantom, "(\mathbf{i})"] & N \arrow[d, Rightarrow, "\widehat{tr_i}"] & & N\_R_j \arrow[lll, "e_j"'] \arrow[d, "e'_j"'] \\
N\_R \arrow[r, "e'_i"'] & \widehat{N_i} \arrow[r, "\widehat{e_i}"'] & \widetilde{N} \arrow[r, "\widetilde{e}"'] & N'
\end{tikzcd}$$

5 Conclusion

In this paper we present a generic component concept capable of handling multiple provide and require interfaces and multiple access. This includes the definition of generic components and a hierarchical composition operation for multiple interfaces. Moreover, we introduce the concurrent application of composition steps, called parallel composition. Based on that we prove the result of two overlapping composition steps to be independent of the construction ordering. This induces that the given reduction semantics of architectures is unique. The generic concept is instantiated to the sample specification technique of elementary nets which is also used for the small web browser suite in order clarify the main ideas.

Dynamic software architectures that use formal techniques are investigated in [6,5]. Many of those approaches use graph transformations for the specification of dynamic changes and reconfigurations. In [32] we have integrated the generic component concept with high-level replacement systems, a categorical generalization of graph transformations. This work can be considered as the technical foundation for the extension of the approach introduced in this paper to dynamic architectures. Since the semantics of architectures is defined by graph transformations we can apply corresponding transformation engines as for example the AGG tool [1] in order to compute the semantics automatically.

As already mentioned in Sect. 3 the handling of multiple accesses in component architectures depends on the used specification technique and the corresponding instance notion. The composition operations presented in this paper are fully adequate for techniques with a loose semantics, i.e. each specification induces a set of valid instances. For techniques with a close semantics there are two possibilities of resolving multiple access. The first variant glues the requiring components over the providing one. This is suitable for a shared access, as used in our example of Sect. 2. Exclusive access requires a different composition operation that creates a copy of the required component for each request. See [26] for details.

References

[1] AGG: A development environment for attributed graph transformation systems. http://tfs.cs.tu-berlin.de/agg.

[2] R. Allen and D. Garlan. The Wright architectural specification language. Technical Report CMU-CS-96-TBD, School of Computer Science, Carnegie Mellon University, 1996.

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

[4] S. Balsamo, M. Bernardo, and M. Simeoni. Performance evaluation at the software architecture level. volume 2804 of *Lecture Notes in Computer Science*, pages 207–258. Springer Verlag, 2003.

[5] J.S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, Queens University, 2004.

[6] J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

[7] C. Choppy, P. Poizat, and J. Royer. Formal specification of mixed components with Korrigan. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference, APSEC'2001*, pages 169–176. IEEE, 2001.

[8] http://www.fiadeiro.org/jose/CommUnity/.

[9] E.P. Dawis. Architecture of an SS7 protocol stack on a broadband switch platform using dualistic petri nets. In *2001 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 323–326, 2001.

[10] E.P. Dawis, J.F. Dawis, and W.P. Koo. Architecture of computer-based systems using dualistic petri nets. In *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*, October 2001.

[11] V.C. de Paula, G.R.B. Justo, and P.R.F. Cunha. Specifying and verifying reconfigurable software architectures. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 21 – 31, 2000.

[12] H. Ehrig, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. Object-Oriented Connector-Component Architectures. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA)*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[13] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A Generic Component Concept for System Modeling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2002.

[14] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A Transformation-Based Component Framework for a Generic Integrated Modeling Technique. *Journal of Integrated Design and Process Science*, 6(4):78–104, 2003.

[15] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A component framework for system modeling based on high-level replacement systems. *Software and System Modeling*, 3(2):114–135, 2004.

[16] H. Ehrig, J. Padberg, B. Braatz, M. Klein, F. Orejas, S. Perez, and E. Pino. A Generic Framework for Connector Architectures based on Components and Transformations. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA)*, volume 108 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[17] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon. Sara (system architects apprentice): modeling, analysis, and simulation support for design of concurrent systems. *IEEE Trans. Softw. Eng.*, 12(2):293–311, 1986.

[18] J. L. Fiadeiro and T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[19] J.L. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. In R. C. Backhouse and J. Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 178–221. Springer, 2003.

[20] D. Garlan. Formal modeling and analysis of software architecture: connectors, and events. In *Formal Methods for Software Architecture*, volume 2804 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.

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

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

[23] L. Grunske. *Strukturorientierte Optimierung der Qualitätseigenschaften von softwareintensiven technischen Systemen im Architekturentwurf.* PhD thesis, Universität Potsdam, 2004.

[24] W. Hasselbring. Component-based software engineering. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 289–305. World Scientific Publishing, New Jersey, 2002. ISBN 981-02-4974-8.

[25] M. Klein. A Component Concept for System Modeling Based on High-Level Replacement Systems. Forschungsbericht 2003/09, Fakultät IV – Elektrotechnik und Informatik, TU Berlin, 2003.

[26] M. Klein. Compatibility constructions for multiple access in generic component architectures. Forschungsbericht 2006/01, Fakultät IV – Elektrotechnik und Informatik, TU Berlin. To appear.

[27] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, Lecture Notes in Computer Science 989, pages 137–153. Springer, 1995.

[28] J. Magee and J. Kramer. Dynamic Structures in Software Architecture. In *Proc. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, 1996.

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

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

[31] J. Padberg. Formale Techniken für die Beschreibung von Software-Architekturen. In R. Reussner and W. Hasselbring, editors, *Handbuch der Software-Architektur.* d-punkt Verlag, 2005. Accepted.

[32] J. Padberg. Integration of the generic component concepts for system modeling with adhesive HLR systems. *EATCS Bulletin*, 87:138–155, 2005.

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

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

[35] P.S. Thiagarajan. Elementary Net Systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets : Central Models and Their Properties*, number 254 in LNCS, pages 26–59. Springer Verlag, 1987.

[36] J. Woodcock and J. Davies. *Using Z.* Prentice-Hall, 1996.