Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 123–151
www.elsevier.com/locate/entcs

Object-Oriented Connector-Component Architectures

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

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

F. Orejas, S. Pérez, E. Pino[2]

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

Abstract

This paper presents an important extension of our contribution to FESCA '04, which presented a generic framework for connector architectures. These architectures were defined by components, consisting of a body specification and a set of export interfaces, and connectors, consisting of a body specification and a set of import interfaces plus connecting transformations in both cases. A major restriction of this framework was given by the assumption of non-overlapping connector interfaces.
In order to make the generic framework for connector architectures more applicable, it is enriched by the possibility of handling overlapping connector interfaces. Fortunately, it is possible to extend the main results presented at FESCA '04 also to the new framework. Moreover, it is shown that the new framework can be applied to UML class diagrams, state machines and sequence diagrams as heterogeneous specification techniques. The resulting connector framework, including a concept for the composition of components and architectural reduction for UML specifications, is illustrated by a case study concerning the meta data management in Topic Maps.

Keywords: Modules and Interfaces, Components and Connectors, Object-Orientation

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

doi:10.1016/j.entcs.2005.02.048

1 Introduction

The importance of architecture descriptions has become most obvious over the last decade (see e. g. [15,16,7,8,6]). Various formalisms have been proposed to deal with the complexity of large software systems. The idea of dividing computation and coordination in software programs and in the corresponding specifications, mainly motivated by Allen and Garlan in [1], found a wide acceptance in today's software engineering and research (see e. g. [17]). In most of these approaches, the division is realized by the use of components as computation units and connectors as coordination units. In our recent paper [4] we presented a generic approach based on [3] to handle this kind of architectures, including a notion of component composition and a semantics that calculated a single component for each architecture. Moreover, we have studied instantiations to formal specification techniques like Petri Nets and CSP. In our approach a component consists of a body and a set of export interfaces, and connections between export and body. A connector consists of a body and a set of disjoint import interfaces. These connections are generic to allow a great variety of instantiations.

This paper has two main aims. The first one is to extend the generic framework by allowing overlapping connector interfaces. The second aim is to apply the new generic framework to object-oriented specification techniques in the sense of UML(see [14]).

In order to reach the first aim, we have to relax the requirements for the parallel extension property, which is used to calculate the composition of components along connectors. The difference with respect to the framework in [4] is that we require a parallel extension of transformations, if all given transformations are compatible with all given embeddings instead of requiring complete independence of the embeddings.

The second aim includes an instantiation of the generic concepts to UML diagrams, where we consider class diagrams, state machines and sequence diagrams in this paper. This requires to define transformations, embeddings, extension and parallel extension for these types of UML diagrams. Compatibility of transformations and embeddings means that all overlapping parts are commonly refined by the given transformations.

The paper is organized as follows. In Section 2 we start with a small case study for an object-oriented component architecture. This is an explicit example of the advanced generic architecture framework presented in Section 3. Based on that, we define the semantics of connector architectures in Section 4. The main result in Section 4 shows existence and uniqueness of architecture semantics, which is based on compatibility of component com-

position in Section 3, within the extended framework allowing overlapping connector interfaces. Section 5 then presents the instantiation of the generic framework to UML diagrams, which is the concrete framework for our case study in Section 2. In Section 6 we conclude with a brief discussion of related work and an outlook to future research.

2 Case Study: Meta data Management in Topic Maps

In this section we will model a small case study concerning the management of meta data using an object-oriented connector-component architecture based on UML. The corresponding architecture framework for UML is an instantiation of the generic framework for architectures presented in Sections 3 and 4. This instantiation will be described in more detail in Section 5.

In our case study we consider an example system for the management of meta data in Topic Maps [9], which is an ISO standard for the 'Semantic Web'. The main notions of Topic Maps are topics and associations between them. For example we want to describe the meta data of music media files. Topics in this area include medium, track, and artist, which are related by associations like the release of a medium by an artist, the containment of a track on a medium, or the production of a track by some performer and composer.

The system shall be able to exchange the data of arbitrary Topic Maps via the HTTP protocol (see [12]). This way it shall be possible to share the data on one hand in a server based fashion, where a powerful web server processes the queries of lots of clients, and on the other hand in a peer-to-peer fashion, where clients exchange data directly. As exchange format the standardized XML transfer syntax [10] for Topic Maps shall be applied.

The music meta data shall also be used to manage media like MP3 files. The system shall be able to change the ID3 meta data (see [13]) of existing MP3 files according to the meta data in the Topic Map and move the file into a media file hierarchy with canonicalized names.

The domain of this example can easily be enhanced to capture other media and their meta data. For example the bibliographic data concerning scientific publications could be modeled this way and used to organize a collection of bookmarks and electronic versions of these publications.

Architecture of the Example

The requirements are specified in an architecture consisting of components and connectors as shown in Figure 1, where we use the package stereotypes <<component>> and <<connector>> and the dependency stereotype <<transform>> to identify the notions of the generic architecture framework.

The <<architecture>> packages in this abstract view correspond to the architecture graphs of our generic framework in Section 3. The components are Ontology, Server, and Manager representing the three main areas of requirements. They are connected via the connectors SrvSrc between Server and the data model in Ontology and ManOnt between Manager and the domain ontology in Ontology.

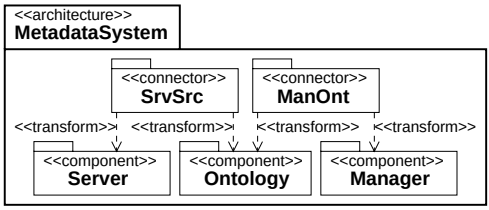

Fig. 1. Architecture of the example

We will use the additional package stereotypes <<componentBody>>, <<componentExport>>, <<connectorBody>> and <<connectorImport>> and the dependency stereotype <<embed>> to describe the substructure of components and connectors according to the generic framework. A component consists of several export packages with transform dependencies to a body package and a connector consists of several import packages with inclusion dependencies to a body package.

Ontology Component

The Ontology component consists of the packages DataModel and MusicOntology and corresponding export packages for both of them. The structure of the component is depicted in Figure 2. The DataModel package shown in Figure 3 specifies a simplified version of the Topic Maps data model described in [11] by a UML class diagram. Additionally two methods are declared which will be used by the Server component below. The first method getByIdentifier of TopicMap takes a URI as parameter and returns a topic containing the given URI as identifier if it exists. This is specified by the OCL constraint for the method. The second method serialize of Topic shall return an XTM

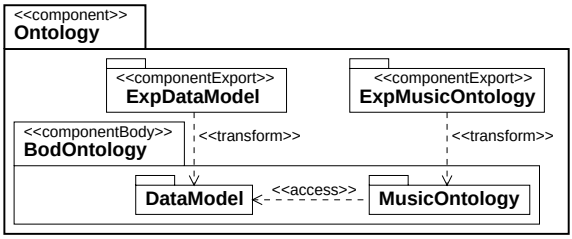

Fig. 2. Component Ontology

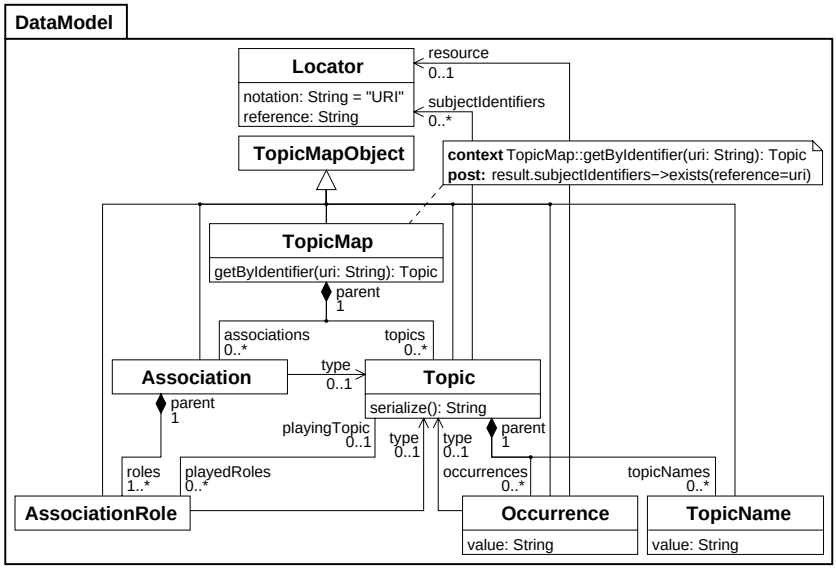

Fig. 3. Data model of the system

serialization [10] of the topic. A requirement not specified in the UML model is the intention that the serialization includes the topic with all non-local occurrences, all topic names, and all associations, where the topic plays an association role. The data model is completely exported in the ExpDataModel export package in Figure 2 in order to be accessible for applications. The domain ontology itself is specified by the class diagram of the MusicOntology package in Figure 4. Topics are specialized by the classes Track, Medium, and Artist, which are related by the associations Containment, Release, and Production. There are attributes for selected names in UTF8 and ASCII encoding in the topic classes, which are required to be included in the topic names of the data model by OCL constraints in the notes attached to the classes in Figure 4. The Track class additionally declares some methods to access the names of it's associates directly and add MP3 files as occurrences to the track. The effects of these methods are again specified by OCL constraints in the notes of the class diagram. Only the class Track and the used File class are exported in ExpMusicOntology in Figure 2, so that applications can access the data through the methods of Track.

Server-Source Connector

To connect the Ontology component with an HTTP server to provide the topic map data to clients we use a generic connector SrvSrc modeling the connection from some data server to it's underlying data source. This connector is given in Figure 5. The data server retrieves the resource for a given URI

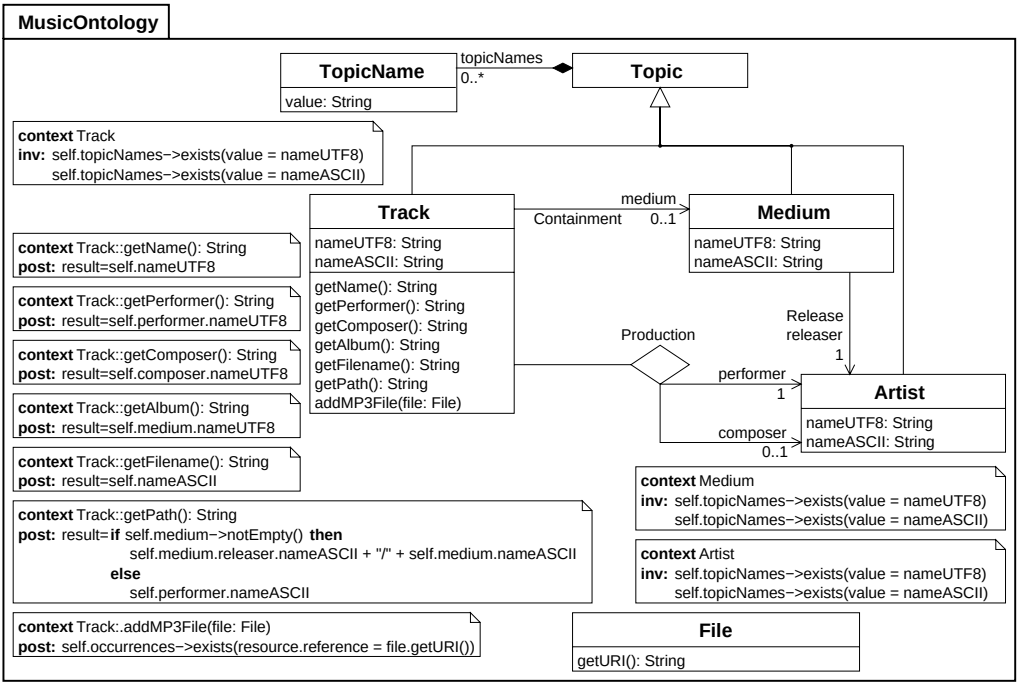

Fig. 4. Ontology of the system

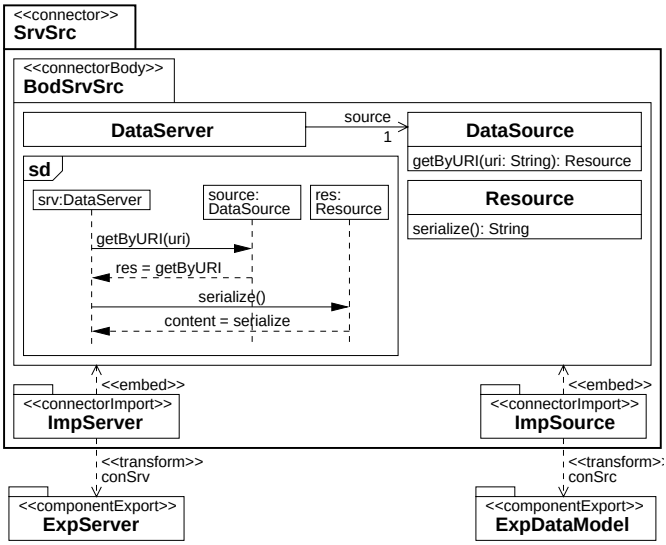

Fig. 5. Connector SrvSrc

from the data source and afterwards serializes it. This sequence is specified by the sequence diagram of the package. The import ImpServer is identical to the connector body, because a data server component will need all entities described in the connector. On the other hand the interface ImpSource contains

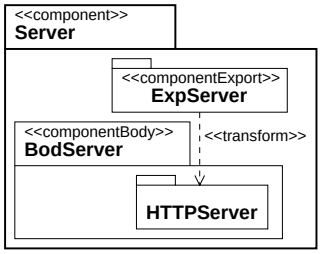

Fig. 6. Component Server

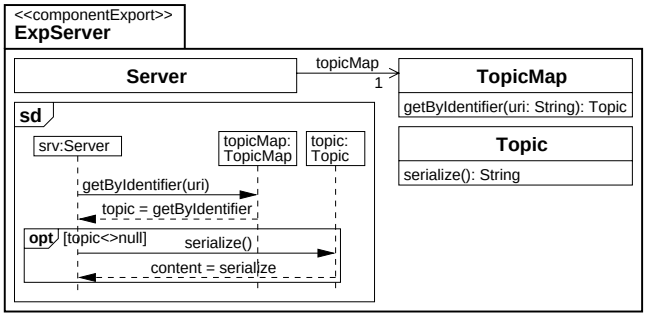

Fig. 7. Export package ExpServer

only the classes DataSource and Resource, since a data source component does not need to know about the server. There is a connection conSrc, which is a <<transform>> dependency, between ImpSource and ExpDataModel renaming the class DataSource with the method getByURI to the class TopicMap with method getByIdentifier and the class Resource to Topic (the method serialize is not renamed). The renamed model is included in ExpDataModel. The connection conSrv to the export of the Server component will be described in the next paragraph.

Server Component

The server component in Figure 6 shall be used to satisfy the ImpServer import of the connector in the previous paragraph. One of the requirements of the generic architecture framework in Section 3 is that overlapping parts of imports are identically transformed. So we have to perform the renamings of conSrc also in conSrv. Moreover we restrict the sequence to only use the serialize method if a topic was found by getByIdentifier. These transformations yield the export of the Server component in Figure 7. This export is further transformed to the model of the internal HTTP server and it's protocol in the HTTPServer package in Figure 8. This server receives HTTP requests and tries to find a topic which has the requested URI as one of it's subject identifiers. It responds with an HTTP response with the serialization of the found topic

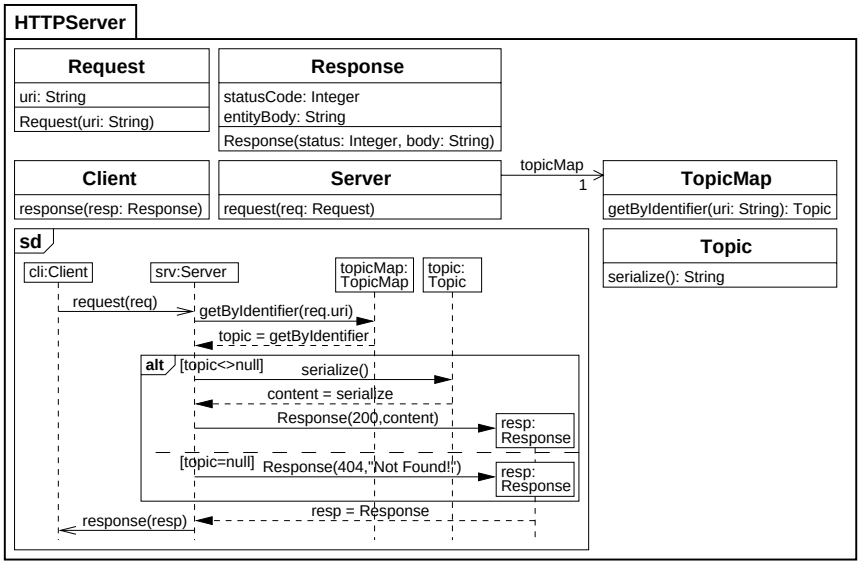

Fig. 8. HTTP server requirements

or a response with status "404 – Not Found" if no topic was found. With this protocol it is possible to ask a server for possible information about a subject identified by some URI. In a peer-to-peer network this could be used to share information about topics among clients knowing topics for the same subject identifier.

Manager-Ontology Connector

In order to connect the ontology to some resource manager, e. g. the MP3 file manager in the next paragraph, we define the connector ManOnt shown in Figure 9. The connector abstractly describes the relation between some manager and a topic being able to describe files, i. e. files can be added as occurrences to the topic and their meta data can be exported as strings. The manager is now required to have a method insertFile which alters some meta data tag inside the file to reflect the data in the topic and then adds the file as occurrence to the topic. This is specified by the state machine in Figure 9. The import ImpOntology just contains FileTopic and File, because these have to be provided by the ontology. This import can be connected via the connection conOnt to the export ExpMusicOntology by choosing Track as FileTopic and refining the getMetadata method to special methods for different kinds of meta data. The import ImpManager contains the whole body of the connector, because managers will access and transform all of it's contents. This import is connected via conMan to the export of the Manager component described in the next paragraph.

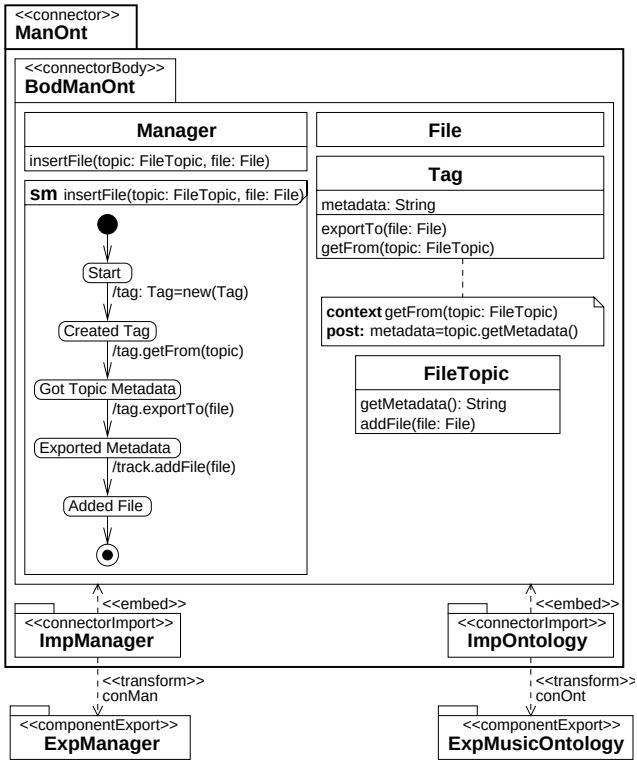

Fig. 9. Connector ManOnt

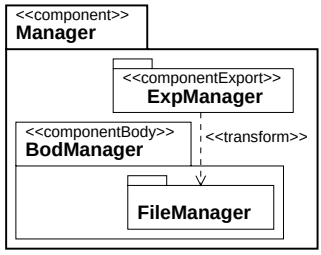

Fig. 10. Component Manager

Manager Component

Figure 10 shows the structure of the component Manager. To instantiate the ImpManager package of the ManOnt connector it is transformed via the connection conMan resulting in the package in Figure 11. Because FileTopic is transformed to Track in ExpMusicOntology this is also done in ExpManager. To reflect this change also for the Tag class it is transformed to ID3Tag which represents the meta data in ID3 tags of MP3 files. The export is then further transformed to the FileManager package in BodManager shown in Figure 12. In this body the requirement of a canonicalized hierarchy of media files is

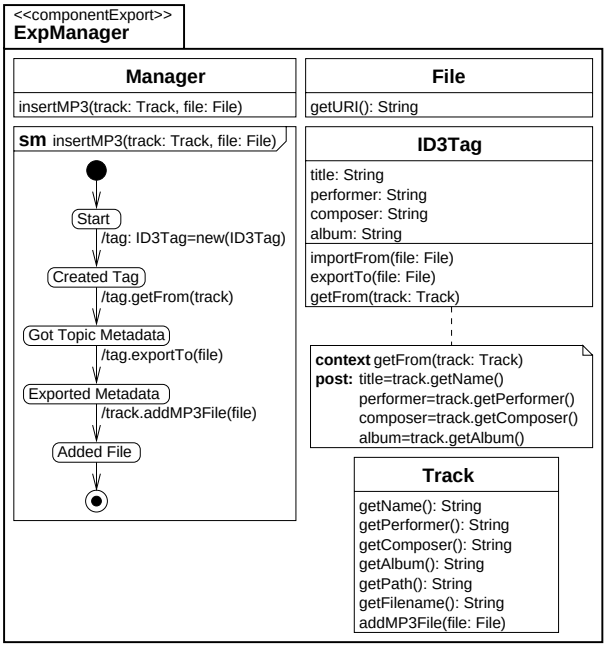

Fig. 11. Export package ExpManager

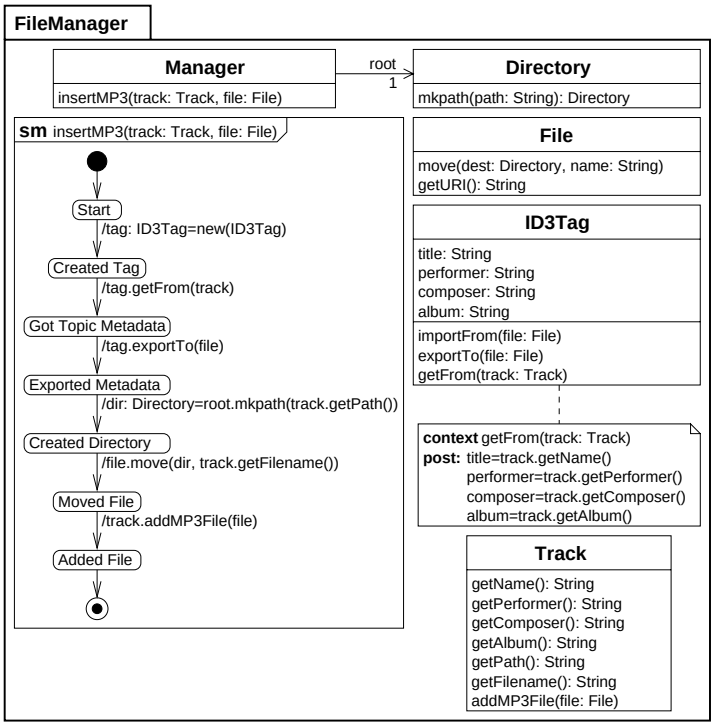

Fig. 12. File management requirements

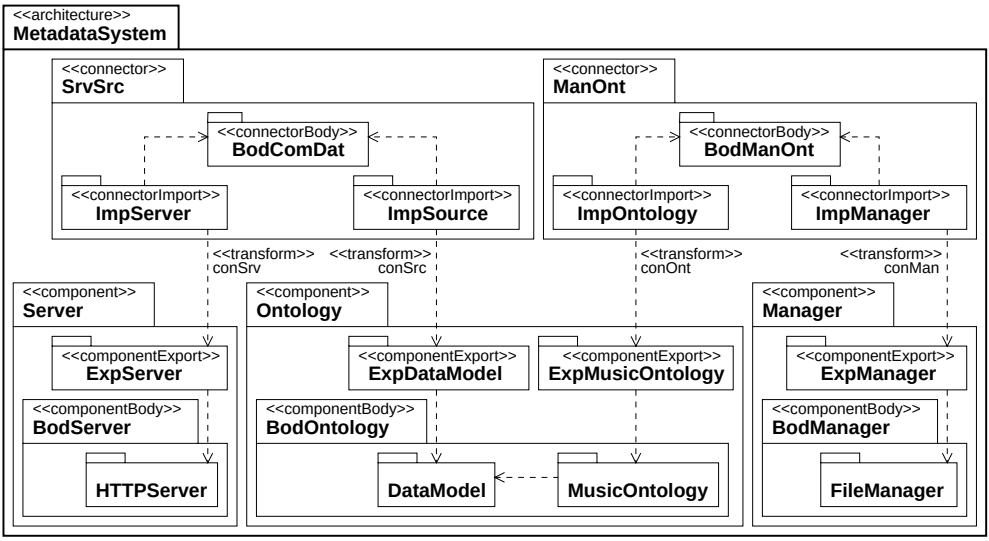

Fig. 13. Detailed architecture of the example

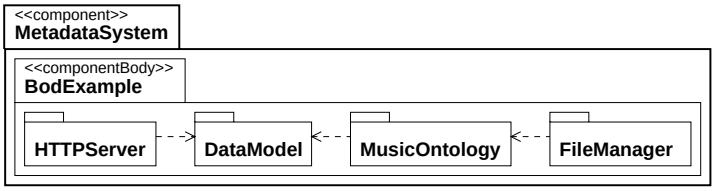

Fig. 14. Architecture semantics of the example

additionally considered by adding a Directory class and a move method for files and enriching the state machine for insertFile.

Detailed Architecture of the Example

With these components we obtain the detailed architecture of the system shown in Figure 13. This detailed architecture corresponds to the architecture diagram as defined in the generic architecture framework in Section 3.

In order to define the semantics of the architecture in Figure 13 the architecture framework demands in Section 4 the ability to flatten such architectures. This is done by applying all transformations simultaneously to get one global body and then forget the connectors and connected exports which is justified by Theorem 3.11. The instantiation of this requirement to UML will be discussed in Section 5. Applying this complete reduction to the example architecture yields the component in Figure 14, where the body is given by the packages DataModel in Figure 3, MusicOntology in Figure 4, HTTPServer in Figure 8, and FileManager in Figure 12 and the dependencies between them induced by the usage of classes in the connector bodies. Note that in this

case we have no export packages left in the resulting component, because all exports have been used by the connectors already.

Such complete reductions of architectures to single components will be defined as the semantics of architectures in Section 4. But first the generic architecture framework itself is formally introduced in the next section.

3 The Generic Architecture Framework

In this section we present a generic framework for connector architectures, which is based on the ideas of our framework presented at FESCA '04 in [4]. The new version in this paper, however, is more flexible, because it allows overlapping connector interfaces in contrast to non-overlapping ones in [4]. The present framework is generic with respect to several parameters. We use a class of specifications (or models, respectively) and classes of corresponding transformations and embeddings between specifications (or models) that can be instantiated to concrete specification (modeling) techniques. We only require the following properties, which have to be ensured by the used concrete specification (modeling) technique, when the framework is instantiated.

- Transformations are closed under composition, i. e. given two transformations $t: SP \Rightarrow SP'$ and $t': SP' \Rightarrow SP''$, then there exists a composed transformation $t' \circ t: SP \Rightarrow SP''$.
- There is a special identity transformation which is neutral with respect to the composition of transformations. This means for each specification SP we have a transformation id_{SP} with $t \circ id_{SP} = t = id_{SP'} \circ t$ for each transformation $t: SP \Rightarrow SP'$.
- Embeddings of specifications have to be closed under composition. Given two embeddings $e_1: SP_1 \to SP_2$ and $e_2: SP_2 \to SP_3$, then we require a composed embedding $e_2 \circ e_1: SP_1 \to SP_3$.
- Analog to the transformations we require identical embeddings id_{SP} with $e \circ id_{SP} = e = id_{SP'} \circ e$.
- Finally, we have to require that embeddings are a special case of transformations, such that the identities are compatible.

For these generic notions of transformations and embeddings we require the following extension and parallel extension property.

Definition 3.1 (Extension Property) Given an embedding $e: SP \to SP_1$ and a transformation $t: SP \Rightarrow SP'$ as depicted in Figure 15, such that e is consistent with respect to t. Then there is a canonical extension diagram (1) with embedding e' and transformation t_1. In the special case of t being also

Fig. 15. Extension diagrams

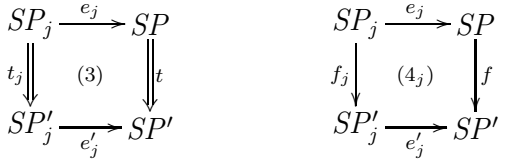

Fig. 16. Parallel Extension Diagrams

an embedding f, we have consistency and a unique mutual extension diagram (2), where f_1 extends f via e and e' extends e via f and (2) commutes.

Note that in diagram (1) we do not require equality of the transformations $e' \circ t$ and $t_1 \circ e$, but only equality of the corresponding domain SP and co-domain SP'_1. The above mentioned consistency between embeddings and transformations is generic in the general framework and can be instantiated differently for each instantiation. In order to handle multiple interfaces we will also need the following parallel extension property.

Definition 3.2 (Parallel Extension Property) Given families of transformations $t_j: SP_j \Rightarrow SP'_j$ and embeddings $e_j: SP_j \to SP$ for indices j in some finite index set J as shown in Figure 16, such that the family of embeddings $(e_j)_{j\in J}$ is consistent with respect to the family of transformations $(t_j)_{j\in J}$, then there is a canonical parallel extension diagram (3) with embeddings e'_j and transformation t, such that:

(i) Parallel extension diagrams are closed under vertical composition.

(ii) If all t_j are embeddings f_j, then we have consistency, the result t is an embedding f and (4_j) commutes for all $j \in J$. If additionally for some $k \in J$ all other t_j with $j \in J \setminus \{k\}$ are identities, then (4_k) is a mutual extension diagram.

Again, the notion of consistency for a family of embeddings w.r.t. a family of transformations is generic and can be instantiated differently for different specification or modeling techniques. Now we are able to define generic components and connectors.

Definition 3.3 (Component) A component $COMP = (B, (e_k)_{k\in K})$ is given by the body B and a family of export interfaces E_k with export transforma-

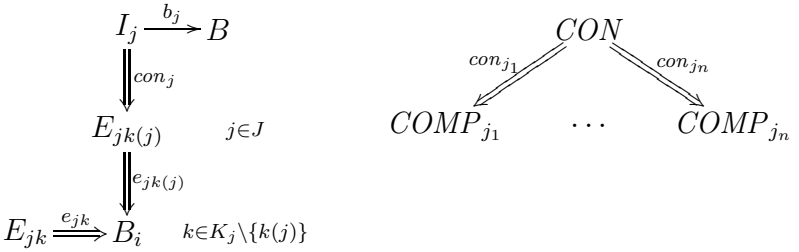

Fig. 17. Connector Diagram and Connector Graph

tions $e_k: E_k \Rightarrow B$ for $k \in K$.

Definition 3.4 (Connector) A connector $CON = (B, (b_j)_{j \in J})$ is given by the body B and a family of import interfaces I_j with body embeddings $b_j: I_j \to B$ for $j \in J$.

Next, we define formally how a connector connects different components.

Definition 3.5 (Connector Diagram and Graph) Given a connector $CON = (B, (b_j)_{j \in J})$ of arity $n = |J|$ and for each $j \in J$ a component $COMP_j = (B_j, (e_{jk})_{k \in K_j})$ with a connector transformation $con_j: I_j \Rightarrow E_{jk(j)}$ with $k(j) \in K(j)$, such that $(b_j)_{j \in J}$ is consistent with respect to $(e_{jk(j)} \circ con_j)_{j \in J}$, then we obtain the *connector diagram* in Figure 17 and the *connector graph* in Figure 17, where the following conditions have to hold:

- A connector diagram consists of n import interface nodes I_j of $n+1$ body nodes B_j and B, and of $\sum_{j \in J} |K_j|$ export interface nodes E_{jk}, even if some of the specifications may be equal, e.g. $B_1 = B_2$.
- Circular connections as in (2) of Figure 18 are forbidden, unless we duplicate the body as in (1) of Figure 18. Otherwise the semantics of such a circular architecture is not defined, as it would cause the identification of the export interfaces $E_{1k(1)}$ and $E_{1k(2)}$ of component $COMP_1$ or other kinds of unwanted side effects.
- Note that the interfaces $b_j: I_j \to B$ for the connector are not required to be disjoint, but they are allowed to overlap.

Next, we carry the concept of overlapping connector interfaces forward to whole architectures of components and connectors. 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.

Definition 3.6 (Architecture diagram) An *architecture diagram* D_A of arity (k, l) is a diagram built up from the l connector diagrams, the k com-

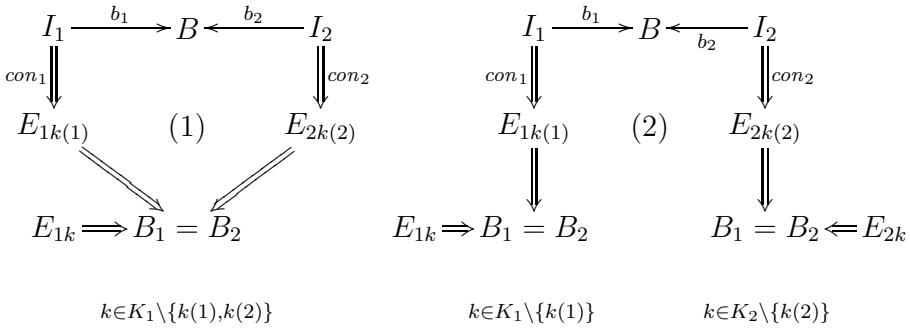

Fig. 18. Non-circular and circular connector diagrams

ponent 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 $con : I \Rightarrow E$, 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 $con : I \Rightarrow E$.

(iii) *Non-circularity:* The architecture diagram D_A is connected and non-circular aside from the arrows' direction.

In order to depict whole architectures clearer we introduce the notion of architecture graphs abstracting from the direct interface connections and only revealing, which components are connected by which connectors.

Definition 3.7 (Architecture graph) An *architecture graph* G_A for an architecture diagram D_A is obtained 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.

Definition 3.8 (Architecture) 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.

Definition 3.9 (Component Composition) The composition by a connector with index set J is defined as follows: Given the corresponding connector diagram (see Figure 17) we construct the parallel extension diagram (1) in Figure 19. The result of the composition of the components $(COMP_j)_{j \in J}$ by the connector CON with the connection transformations $(con_j)_{j \in J}$ is again a component $COMP = (B', (e'_{jk} : E_{jk} \Rightarrow B')_{(j,k) \in J \otimes K})$ with $J \otimes K = \{ (j, k) \mid j \in$

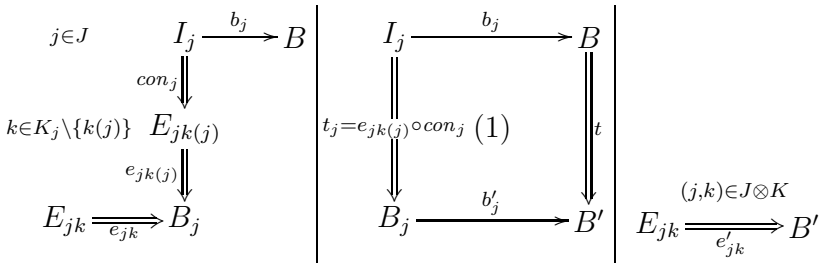

Fig. 19. Composition

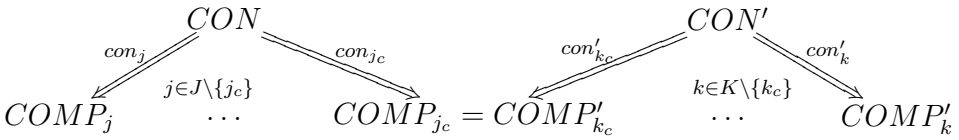

Fig. 20. General connectors and components

$J, k \in K_j \setminus \{k(j)\}\}$ and $e'_{jk} := b'_j \circ e_{jk}$ for all $(j,k) \in J \otimes K$. In this case we say that e'_{jk} are extensions of e_{jk} (See Figure 19). 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

$$COMP = CON((COMP_j)_{j \in J}, (con_j)_{j \in J}) \ .$$

The next theorem states that the result of two overlapping compositions via two connectors is independent of the order the single compositions are calculated.

Theorem 3.10 (Compatibility of Component Composition) *Given an architecture A with arbitrary components and two connectors with the architecture graph G_A in Figure 20, then we have the equality of the following three expressions:*

$$(E1)\ CON((COMP_j)_{j \in J \setminus \{j_c\}}, CON'((COMP'_k)_{k \in K}))$$

$$(E2)\ CON'(CON(COMP_{j \in J}), (COMP'_k)_{k \in K \setminus \{k_c\}})$$

$$(E3)\ CON + CON'((COMP_j)_{j \in J}, (COMP_k)_{k \in K})$$

where (E1) and (E2) are different sequential compositions and (E3) is a parallel composition of the components via the two connectors.

In the following we consider the special case of binary connectors and components with two export interfaces each. The proof of this special case shows how to use the extension and the parallel extension property and can be extended without problems to the general case of Theorem 3.10 which is needed in the proof of Theorem 4.5.

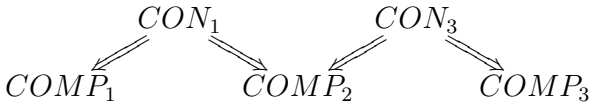

Fig. 21. Binary connectors and components (G_A)

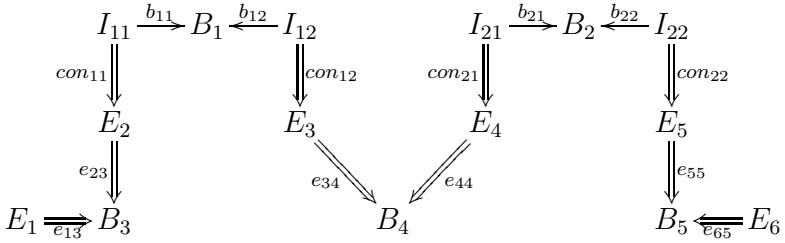

Fig. 22. Architecture diagram D_A

Theorem 3.11 (Associativity of Binary Component Composition) *Given an architecture A with binary components and binary connectors with the architecture graph G_A in Figure 21, then we have the following associativity law:*

$$(E1)\ (COMP_1 +_{CON_1} COMP_2) +_{CON_2} COMP_3 =$$
$$(E2)\ COMP_1 +_{CON_1} (COMP_2 +_{CON_2} COMP_3) =$$
$$(E3)\ COMP_1 +_{CON_1} COMP_2 +_{CON_2} COMP_3,$$

where the last expression corresponds to a parallel composition explained below.

Proof. Let us consider the architecture diagram D_A in Figure 22 corresponding to the architecture graph G_A given above. We first present the parallel composition corresponding to the result of expression (E3). Note, that the embeddings (b_{11}, b_{12}) are consistent with respect to the transformations $(e_{23} \circ con_{11},\ e_{34} \circ con_{12})$ according to the definition of connector diagrams (see Definition 3.5), which allows to construct the parallel extension diagram in the left part of Figure 23 with transformation $t_1\colon B_1 \Rightarrow B_1'$ and embeddings $b_0\colon B_3 \to B_1'$ and $b_1'\colon B_4 \to B_1'$. For similar reasons we obtain the parallel extension diagram in the right part of Figure 23 with transformation $t_2\colon B_2 \Rightarrow B_2'$ and embeddings $b_2'\colon B_4 \to B_5$ and $b_5\colon B_5 \to B_2'$. From the embeddings b_1' and b_2' we construct the mutual extension diagram (1) in Figure 23, where we do not need an additional consistency condition by the extension property (see Definition 3.1). The composition $(COMP_1 +_{CON_1} COMP_2) +_{CON_2} COMP_3$ in expression (E1) corresponds to the diagram in Figure 24, where the result of $COMP_1 +_{CON_1} COMP_2$ is given by $(B_1', E_1 \Rightarrow B_3 \xrightarrow{b_0} B_1', E_4 \Rightarrow B_4 \xrightarrow{b_1'} B_1')$ by

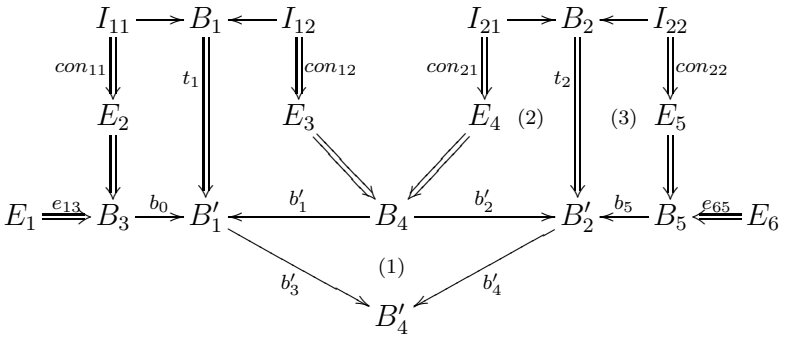

Fig. 23. Parallel composition

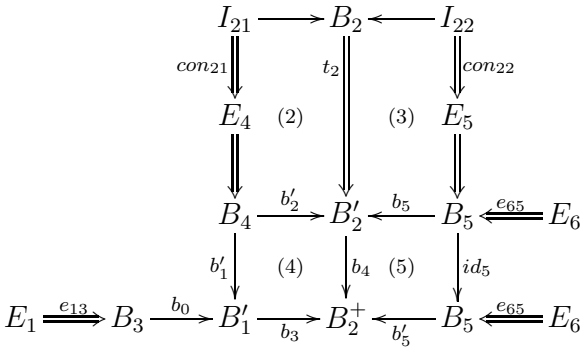

Fig. 24. Stepwise composition

Definition 3.9. Now we consider in Figure 24 the same parallel extension diagram (2,3) as above in Figure 23 and a new one by (4,5) according to part (ii) of the parallel extension property (see Definition 3.2). Using part (i) the vertical composition property implies that (2+4, 3+5) is a parallel extension diagram leading to the bottom line in Figure 24 as result of expression (E1). But (4,5) is a special case of a parallel extension diagram with identity, such that, according to (ii) of Definition 3.2, (4) becomes a mutual extension diagram of b'_1 by b'_2 and hence equal to (1) above. This implies $B'_4 = B_2^+, b'_3 = b_3, b'_4 = b_4$ and $b_4 \circ b_5 = b'_5$ by (5). This implies that the result of expression (E1), given by $(B'_4, E_1 \stackrel{e_{13}}{\Longrightarrow} B_3 \stackrel{b_0}{\to} B'_1 \stackrel{b'_3}{\to} B'_4, E_6 \stackrel{e_{65}}{\Longrightarrow} B_5 \stackrel{b_5}{\to} B'_2 \stackrel{b'_4}{\to} B'_4)$, and of expression (E3), given by the bottom line of the diagram in Figure 24, are equal. The dual argument shows that the result of expressions (E2) and (E3) are equal, where diagram (1) has to be considered as extension of b'_2 by b'_1. But this is appropriate, because we have assumed in the general framework that (1) is a mutual extension diagram. □

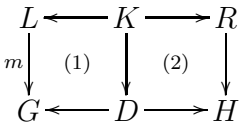

Fig. 25. $G \Rightarrow H$

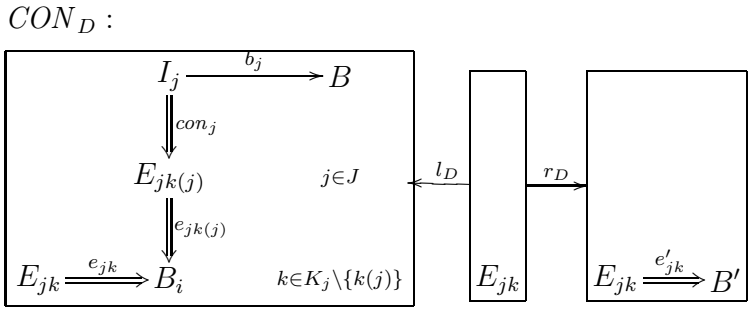

Fig. 26. Diagram reduction rule

4 Semantics of Architectures

In this section we define the semantics of architectures. In fact, we show that we can 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, more precisely the *Double Pushout* approach (see [2]). In fact, a derivation step in this approach is given by two pushout diagrams (1) and (2) in Figure 25, written $G \Rightarrow H$ 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), and H is the gluing of D and R along K in (2), respectively.

Definition 4.1 (Diagram Reduction Rule) Given an architecture A with the architecture diagram D_A there is for each connector CON the *diagram reduction rule* CON_D, as depicted in Figure 26, where B' and $e'_{jk} = b'_j \circ e_{jk}$ is defined by the composition:

$$\begin{aligned} COMP &= CON((COMP_j)_{j\in J}, (con_j)_{j\in J}) \\ &= (B', (e'_{jk}\colon E_{jk} \Rightarrow B')_{(j,k)\in J\otimes K}) \end{aligned}$$

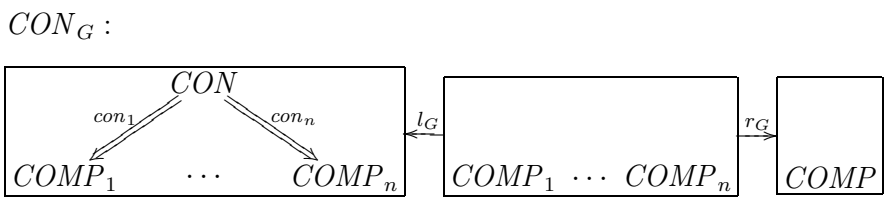

Fig. 27. Graph reduction rule

Definition 4.2 (Graph Reduction Rule) Given an architecture A with the architecture graph G_A. The corresponding *graph reduction rule* CON_G is shown in Figure 27, where $COMP_1, \ldots, COMP_n$ are mapped to $COMP$.

A reduction step $CON_D: D_A \Rightarrow D_{A'}$ and $CON_G: G_A \Rightarrow G_{A'}$, respectively, is given by a derivation step in the *Double Pushout* approach to graph transformations at the level of architecture diagrams or architecture graphs, respectively. For both derivation steps we have inclusions for the matches. Note that although r_G is neither injective nor label-preserving, the labels of $G_{A'}$ for the reduction rule $CON_G: G_A \Rightarrow G_{A'}$ are well-defined by G_A and $COMP$, nevertheless.

Definition 4.3 (Architecture Reduction Rule) An *architecture reduction rule* for a given architecture A is a tuple $CON = (CON_D, CON_G)$ given by a diagram reduction rule CON_D for the architecture diagram D_A and a corresponding graph reduction rule CON_G for the architecture graph G_A.

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.

Given an architecture A consisting of k components and l connectors and a corresponding architecture reduction rule $CON = (CON_D, CON_G)$ we obtain reductions $CON_D: D_A \Rightarrow D_{A'}$, $CON_G : G_A \Rightarrow G_{A'}$ and $CON: A \Rightarrow A'$, where A' is a new architecture with $k-n+1$ components, $l-1$ connectors, architecture diagram $D_{A'}$ and architecture graph $G_{A'}$.

The corresponding proof will be presented in [5]. Now we can give the *semantics of an architecture* as the result of as many reduction rules as possible.

Definition 4.4 (Architecture semantics) The semantics of an architecture A is any component $COMP$ obtained by a sequence of architecture re-

duction steps from A to $COMP$,

$$A \Rightarrow^* COMP.$$

The main result given in Theorem 4.5 shows that this semantics always exists and is unique.

Theorem 4.5 (Exist. 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:*

$$A \Rightarrow^* COMP$$

Proof Idea. This theorem uses the fact that the presented reduction rules satisfy the Church-Rosser property, i. e. the result of a sequence of reduction steps is independent from the order of the steps. This can be shown using a well known local Church-Rosser property for independent graph transformations (see [2]), which are independent reduction steps in our case. For the case of dependent reduction steps we need Theorem 3.10. The result of the reduction sequence is well-defined and unique, since the maximal number of necessary reduction steps is given by the number of connectors and the order of calculation is not relevant. □

The full proof of this theorem will be given in the report [5].

5 Instantiation to UML Models

In this section we will show, how the abstract connector framework can be applied to UML diagrams. In this paper, we regard only the concrete graphical representation of UML diagrams on a more or less intuitive level. This implies that also our instantiation can be given only on an intuitive level. In later stages of our research we want to deal with the corresponding meta-model instances as formal abstract syntax, which would enable us to give a much more detailed definition of connector architectures for UML diagrams. Moreover, we could respect the syntactical dependencies between different diagrams, e. g. the case that a state machine refers to a certain method defined in the class diagram, which are documented in the UML meta-model instances.

5.1 Transformations and Embeddings of UML Diagrams

We will consider (restricted versions) of the following diagram types: class diagrams, state machines and sequence diagrams. For a definition of these diagrams we refer to UML (see [14]).

We allow to attach state machines to classes only. This implies that each state machine SM refers to a corresponding class diagram $cd(SM)$ defining the methods that can be used to label transitions. A sequence diagram SD is also attached to some class diagram $cd(SD)$ defining the classes for all object nodes and the methods used by the message edges. In the first step of the instantiation we will define transformations and embeddings for each of our techniques. In the case study in Section 2 these two notions of connections between UML specifications are referred to as package dependency relations stereotyped by <<transform>> and <<embed>>, respectively.

- A transformation of class diagrams $t_{CD}: CD \Rightarrow CD'$ is given by a mapping of each class $cd \in CD$ to a class $cd' \in CD'$, where the image cd' has to offer at least the functionality of cd up to consistent signature renaming. This means for example, that an attribute $number : Nat$ of CD may be translated to an attribute $number : Int$, if all other occurrences of $number : Nat$ in the class diagram CD are translated to $number : Int$. Of course, the images of the classes are allowed to have additional functionality with respect to their preimage. All connections between classes have to be transformed to corresponding connections of the same type, e. g. associations have to be mapped to associations. Again we allow a renaming of the inscriptions of the connections.
- A transformation of a state machine $t_{SM}: SM \Rightarrow SM'$ requires a transformation of the related class diagram first. This transformation is used to translate the labels of the transitions of the state machine. For each state $st \in SM$ we require an image $st' \in SM'$, which may be renamed. This condition is also required for the state transitions, whose labels have to be transformed in accordance with the transformation of the related class diagram. We allow that the target state machine SM' adds new states and transitions, but we require that all accepted traces of SM are also accepted by the 'enriched' state machine SM', after they have been translated according to the class diagram transformation.
- In the case of sequence diagrams, we consider transformations of sets of diagrams (where each of them represent a possible scenario) instead of single ones. Transformations $t_{SD}: SD \Rightarrow SD'$ are defined in three steps. First, as in the previous case, a transformation of the related class diagram is needed to translate the labels of the interactions in the diagram. The second step is to replace lifelines by disjoint sets of lifelines (including the given lifeline) and interactions (l, m, l') (where l and l' are lifelines and m is a message sent from l to l') by sets of diagrams involving only the lifelines included in the refinements of l and l'. Finally, each diagram in SD must be included in a diagram in SD'.

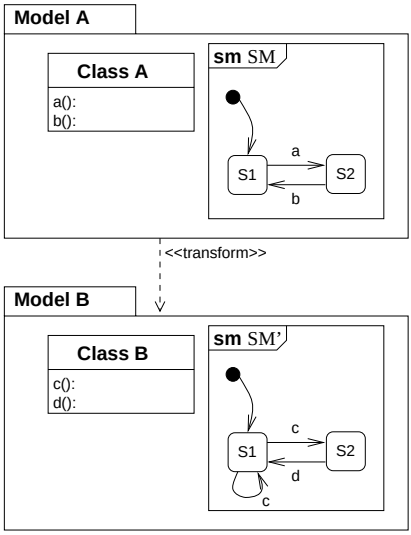

Fig. 28. Sample Transformation of State Machine and Class Diagram

The idea is that when refining a sequence diagram, we may refine lifelines and interactions. In particular, the refinement of a lifeline may involve other lifelines which are considered hidden at a higher level of abstraction. On the other hand, a simple interaction may be replaced by a more complex interaction represented by a set of diagrams.

Now we define embeddings of class diagrams, state machines and sequence diagrams. For sake of simplicity embeddings are inclusions in this paper, which do not allow any renamings. Thus, we are enabled to define the following extension constructions as unions of sets. In both cases, embeddings and transformations, the target diagram is allowed to have additional elements.

Figure 28 shows a transformation of a state machine SM and the related class diagram $cd(SM) = CD$. The transformation of the methods in the sample class, which is not shown explicitly in the figure, renames the methods $a()$: to $c()$: and $b()$: to $d()$:. Since there are no designated final states, the state machine SM accepts $(ab)^*, (ab)^*a$. This is translated along the transformation to $(cd)^*, (cd)^*c$, which is a part of the accepted traces of SM'. Thus, this sample transformation fulfills our requirements for diagram transformations stated above.

5.2 Extension of Diagram Transformations

In the next step of the instantiation we have to verify the extension property (see Definition 3.1), i. e. to define the extension of transformations along embeddings.

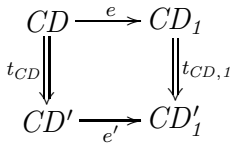

Fig. 29. Extension of Class Diagram Transformations

- For class diagrams consider Figure 29, where three class diagrams CD, CD' and CD_1 are given, connected by a transformation $t_{CD}\colon CD \Rightarrow CD'$ and an embedding $e\colon CD \to CD_1$. The transformation arrows $CD \overset{t_{CD}}{\Longrightarrow} CD'$ and $CD_1 \overset{t_{CD,1}}{\Longrightarrow} CD'_1$ in Figure 29 represent a UML dependency relation of the corresponding packages which is stereotyped as <<transform>>.

  The extension of t_{CD} along e as depicted in Figure 29 is now constructed as follows: We define CD'_1 by adding CD_1 without CD, written $CD_1 \setminus CD$, to the result CD' of the transformation t_{CD}. Note that $CD_1 \setminus CD$ is constructed by removing all classes and class relations from CD_1 that are also part of CD and thus, embedded by e. This may cause ill formed class relations since their targets might have been removed. The well-formedness is restored in CD'_1, since the loose ends of the class relations are connected to the t_{CD} images of the deleted classes.

  We obtain the extension $t_{CD,1}$, because the elements of CD_1 are either directly included to CD'_1 or their renamed versions are taken from CD'. By taking the renaming of the latter elements and the identical embedding of the former we can construct a renaming transformation. Moreover, we obtain the embedding $e'\colon CD' \to CD'_1$, since CD' is a part of CD'_1.

  Note that the construction as described above does only work, if we do not have any name clashes between $CD_1 \setminus CD$ and CD'. We could drop this constraint by defining the construction of CD'_1 by a pushout construction, which avoids name clashes by suitable renaming.
- The extension of state machine transformations works analogous. Given three state machines SM, SM', and SM_1 connected by a transformation $t_{SM}\colon SM \Rightarrow SM'$ and an embedding $s\colon SM \to SM_1$. Since we required related class diagrams for the state machines we can calculate the corresponding class diagram extension. Then we add $SM_1 \setminus SM$ to SM'. This construction ensures that SM'_1 accepts all valid traces of SM_1, because no transitions are deleted.
- In the case of sets of sequence diagrams, we consider that a set SD is embedded in SD' if every diagram in SD is included in some diagram in SD'. Now, if SD is embedded in SD_1 and $t_{SD}\colon SD \Rightarrow SD'$ is a transformation, we define the extension $t_{SD,1}\colon SD \Rightarrow SD'_1$ as follows. First, we replace all the labels in SD_1 by the corresponding labels according to the transformation

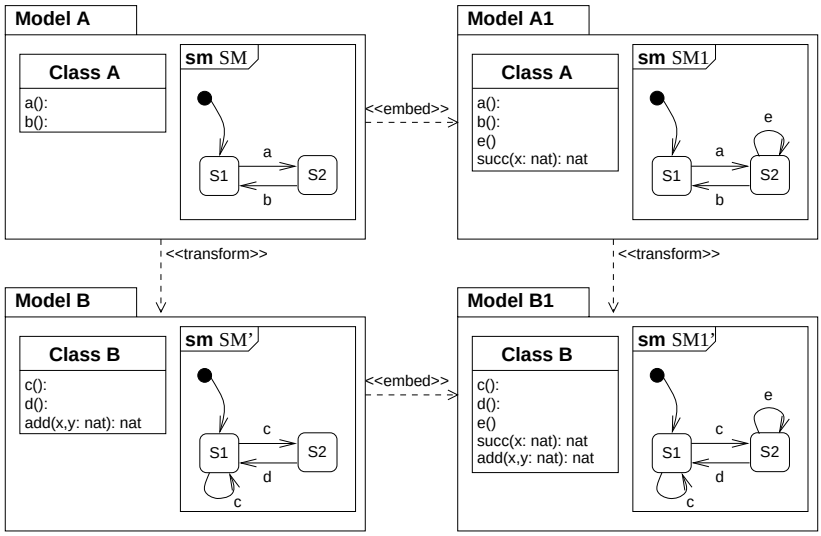

Fig. 30. Sample Extension of State Machine and Class Diagram

of the associated class diagrams. Then we replace all the lifelines in SD_1 by their corresponding refinement according to t_{SD} (if a lifeline is not in SD we assume that it's refinement is the lifeline itself). Finally, we replace all the interactions by their corresponding refinements according to t_{SD} (again, if an interaction is not in SD we assume that it's refinement is the diagram consisting just of that interaction). It may be proven that this construction ensures that there is a transformation from SD to SD'_1 and an embedding from SD_1 to SD'_1.

Figure 30 shows the extension of a state machine transformation and the corresponding class diagram transformation. Intuitively, the transformation $\mathsf{SM} \Longrightarrow \mathsf{SM}'$, which renames the transition labels a to c and b to d and adds a new transition labeled with c, and it's corresponding class diagram transformation are applied to the state machine SM1. This means, the diagram elements embedded from the state machine SM1 and it's corresponding class diagram, are replaced by their images w. r .t. the transformation $\mathsf{SM} \Longrightarrow \mathsf{SM}'$. The remaining elements in the state machine SM1 and the related class diagram are copied unchanged. Finally, all new elements of SM', e. g. the method add, are added.

In the final step of the instantiation we have to verify the parallel extension property (see Definition 3.2), i. e. to define the parallel extension of transformations. Let class diagrams CD and CD_j with transformations $t_{CD,j}\colon CD_j \Rightarrow CD'_j$ and embeddings $e_j\colon CD_j \to CD$ for $j \in J$ be given. For the consistency of the families $(e_i)_{i\in I}$ of embeddings and $(t_i)_{i\in I}$ of transformations, we require that $(e_i)_{i\in I}$ is compatible with $(t_i)_{i\in I}$, which means that all overlappings in CD

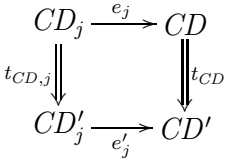

Fig. 31. Parallel Extension of Class Diagrams

with respect to $(e_j)_{j\in J}$ are commonly transformed by $(t_i)_{i\in I}$, i. e. if an element $c \in CD_{j_1} \cap CD_{j_2}$ is in the intersection of two class diagrams, it's image with respect to both transformations, t_{CD,j_1} and t_{CD,j_2}, has to be the same in $CD'_{j_1} \cap CD'_{j_2}$.

The result CD' of the parallel transformation t_{CD} is constructed as follows. First, we join all CD'_j for $j \in J$ to a single class diagram. In the next step we add $CD \setminus \bigcup_{j\in J} CD_j$ to the previous result.

The parallel transformation t_{CD} is then defined as follows. If an element c is an image of any $c_j \in CD_j$ then select the image of c_j with respect to the transformation t_j. This selection is well defined since we required common transformations of the overlappings. Otherwise, i. e. there is no $c_j \in CD_j$ with $e_j(c_j) = c$, c remains unchanged by the constructed parallel transformation t_{CD}. The embeddings e'_j for $j \in J$ are directly induced by the construction of CD'.

In order to avoid name clashes in CD' we require that $CD \setminus \bigcup_{j\in J} CD_j$ is disjoint to $\bigcup_{j\in J} CD'_j$. As discussed above, this could be avoided by constructing CD' by a suitable colimit construction.

The parallel extension property of state machines and sequence diagrams is verified in a similar way.

Summarizing we obtain the following result.

Fact 5.1 (Architecture Framework for UML Models) *Restricted class diagrams, state machines and sequence diagrams as considered above together with the corresponding notions of transformations and embeddings are satisfying the extension property (Definition 3.1) and the parallel extension property (Definition 3.2) of the generic architecture framework in Section 3.*

Proof Idea. The construction of extension and parallel extension diagrams has been discussed already above. It remains to show the properties. First of all, all embeddings preserve the type of the diagram elements and they do not change any inscriptions. Hence, they are special cases of the defined transformations. Embeddings and transformation are closed under composition and the extension diagram of two embeddings $e : SP \to SP_1$ and $f : SP \to SP'$

(with $SP_1 \cap SP' = SP$ for the simplified construction) is given by the union $SP'_1 = SP_1 \cup SP'$ and embeddings $e' : SP' \to SP'_1$ and $f'' : SP_1 \to SP'_1$, which leads to a mutual extension diagram. The construction of parallel extension diagrams above implies that they are closed under vertical composition. Moreover, if all transformations t_i are embeddings, then also t is an embedding and if in addition all $t_2, \ldots, t_n$ are identities, then 3_1 in Figure 15 is a mutual extension diagram, because embeddings are diagram inclusions and do not merge any elements (and they do not change any inscriptions in our simplified version). □

This allows to apply the generic architecture framework to UML models, leading to the concept of components, connectors and architecture diagrams and graphs, architectures, component composition and semantics of architectures as presented in Sections 3 and 4 for UML models. Especially, we obtain the main results "Compatibility of Component Composition" (Theorem 3.10) and "Existence and Uniqueness of Architecture Semantics" (Theorem 4.5) for the UML models considered above.

6 Conclusion

In this paper we have presented object-oriented connector-component architectures of a subset of UML diagrams. More precisely, we have extended our generic framework for connector architectures presented at FESCA '04 (see [4]) to the case of overlapping connector interfaces which allows to apply it to class diagrams, state machines and sequence diagrams with suitable restrictions. In the extended general framework we are able to show as main result compatibility of component composition as well as existence and uniqueness of architecture semantics. The third main result shows that this framework can be instantiated to UML diagrams as discussed above using suitable notions of transformations and embeddings. This allows to apply the generic results to these UML diagrams in general and to an object-oriented connector-component architecture for a meta data management system as a case study in this paper.

The component concept of the UML 2.0, as well as most programming language oriented component approaches, is orthogonal to our approach in the following sense. In contrast to our approach, UML 2.0 components as presented in [14] are intended to describe the distribution of executable program pieces. Our approach is concerned with the structuring of the specification of system requirements and system design. Thus, each of our components might be realized by a set of these software components. Though it might be possible to understand UML 2.0 components as a special case of our approach.

The further examination of this relation would yield a formal foundation for several parts and application scenarios of the UML 2.0 component notion.

The approach in this paper is based on an intuitive graphical representation of UML diagrams. In future work we want to deal with the corresponding meta-model instances as formal abstract syntax, which would allow a much more detailed discussion of the instantiation, and we will also consider more general notions of transformations and embeddings. Moreover, it is possible to consider other UML techniques. Especially with respect to the example in Section 2 it seems sensible to include UML profiles to the components. Ontologies could then be modeled as class hierarchies with respect to an ontology profile. On the other hand profiles could also be used for platform specific implementation models.

In view of system evolution, as for example in the sense of [18], it seems promising to extend the presented framework by means to transform or refine, respectively, not only specifications but whole components and connectors, and thus also by transformations and refinements of component architectures. Refinements of components and connectors can be necessary for different reasons. For example, if a company adds any requirements to their product specification in the middle of the specification process, the developers might have to adapt component interfaces to meet the new requirements. But component refinements should preserve the mutual dependencies with related connectors and concerned components. Such architecture refinement concepts could also be used to formalize the steps between different stages in a defined software development process. With respect to the example in Section 2 this could mean to refine the given architecture by an implementation in Java or .NET.

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.
We would like to thank the anonymous referees for their valuable comments and suggestions for improvements of this paper.

References

[1] Allen, R. and D. Garlan, *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology (1997).

[2] Ehrig, H., *Introduction to the Algebraic Theory of Graph Grammars (A Survey)*, in: V. Claus, H. Ehrig and G. Rozenberg, editors, *Graph Grammars and their Application to Computer Science and Biology*, Lecture Notes in Computer Science **73** (1979), pp. 1–69.

[3] Ehrig, H., F. Orejas, B. Braatz, M. Klein and M. Piirainen, *A Generic Component Concept for System Modeling*, in: R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE '02)*, Lecture Notes in Computer Science **2306** (2002), pp. 33–48.

[4] Ehrig, H., 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 Architecture (FESCA '04)*, Electronic Notes in Theoretical Computer Science **108** (2004), pp. 53–67.

[5] Ehrig, H., J. Padberg, B. Braatz, M. Klein, F. Orejas, S. Perez and E. Pino, *A Generic Framework for Connector-Component Architectures*, Forschungsbericht, Fakultät IV – Elektrotechnik und Informatik, TU Berlin (2005), to appear.

[6] Garlan, D., R. Monroe and D. Wile, *Acme: An Architecture Description Interchange Language*, in: *Proc. of CASCON '97*, 1997, pp. 169–183.

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

[8] Hofmeister, C., R. Nord and D. Soni, "Describing Software Architecture in UML," Kluwer Academic Publishers, 1999 pp. 145–159.

[9] International Organization for Standardization, "Topic Maps," Second edition (2002), ISO/IEC 13250, ISO/IEC JTC 1/SC34 N0322 available from http://www.y12.doe.gov/sgml/sc34/document/0322.htm, last accessed on February 11, 2005.

[10] International Organization for Standardization, "The XML Topic Maps Syntax (XTM 1.1)," (2003), ISO/IEC 13250-3, ISO/IEC JTC1/SC34 N0398 available from http://www.y12.doe.gov/sgml/sc34/document/0398.htm, last accessed on February 11, 2005.

[11] International Organization for Standardization, "Topic Maps – Data Model," (2003), ISO/IEC 13250-2, ISO/IEC JTC1/SC34 N0443 available from http://www.y12.doe.gov/sgml/sc34/document/0443.htm, last accessed on February 11, 2005.

[12] Internet Engineering Task Force, "Hypertext Transfer Protocol – HTTP/1.1," (1999), available from http://www.ietf.org/rfc/rfc2616.txt, last accessed on February 11, 2005.

[13] Nilsson, M., "ID3 tag version 2.4.0," id3.org (2000), available from http://www.id3.org/develop.html, last accessed on February 11, 2005.

[14] Object Management Group, "Unified Modeling Language – Version 2.0 (UML 2.0)," (2004), available from http://www.omg.org/, last accessed on February 11, 2005.

[15] Shaw, M., 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** (1995), pp. 314–315.

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

[17] Wermelinger, M. and J. L. Fiadeiro, *Connectors for Mobile Programs*, IEEE Transactions on Software Engineering **24** (1998), pp. 331–341.

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