Traceable and information-preserving composition of structural models

Nelis Boucké and Danny Weyns and Tom Holvoet

Report CW 533, January 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Traceable and information-preserving composition of structural models

Nelis Boucké and Danny Weyns and Tom Holvoet

Report CW 533, January 2009

Department of Computer Science, K.U.Leuven

Abstract

A structural view is a base view for describing software architectures. A structural view typically comprises several structural models that highlight particular aspects of the software. From our experience, managing the dependencies between structural models is a challenging, yet crucial task for maintaining consistency of an architectural description. In this paper, we focus on the composition of structural models and managing dependencies between models in such a composition. We formally define three basic relations between structural models and a composition operator that enables integration of several structural models based on these relations. We elaborate on several information preserving properties of the composition operator, including traceability, consistency, and completeness. The relations and composition operator have been integrated in the xADL language and the AchStudio tool. We use excerpts of a distributed video-on-demand system designed with the tool as an illustrative case.

Keywords : software architecture, composition, relation, structural model, formal proof
CR Subject Classification : I.2.11, I.2.8, I.2.1

Traceable and information-preserving composition of structural models

Nelis Boucké, Danny Weyns, and Tom Holvoet

DistriNet Labs, K.U.Leuven, Belgium
{nelis.boucke,danny.weyns,tom.holvoet}@cs.kuleuven.be

Abstract. A structural view is a base view for describing software architectures. A structural view typically comprises several structural models that highlight particular aspects of the software. From our experience, managing the dependencies between structural models is a challenging, yet crucial task for maintaining consistency of an architectural description. In this paper, we focus on the composition of structural models and managing dependencies between models in such a composition. We formally define three basic relations between structural models and a composition operator that enables integration of several structural models based on these relations. We elaborate on several information preserving properties of the composition operator, including traceability, consistency, and completeness. The relations and composition operator have been integrated in the xADL language and the AchStudio tool. We use excerpts of a distributed video-on-demand system designed with the tool as an illustrative case.

1 Introduction

The architecture of a software system defines its essential structures, which comprise software elements, the externally visible properties of those elements and the relationships between them [4]. A structural view is a base view for any architectural description. A structural view typically comprises several structural models that highlight particular aspects of the software. Since the models of a structural view describe parts of the same system there are dependencies among the models. From our experience with designing non-trivial architectures, managing such dependencies remains a challenging, yet crucial task for maintaining consistency of an architectural description. During architectural design of several realistic case studies, including an industrial transportation system for logistic services [19][9] and a traffic control system [15], we encountered the following problems:

- Structural models often overlap, e.g. the same element appears in several models possibly in different roles; several elements overlap because a component in a model is refined in another model; or a number of models overlap with another model that describes an overview. Refraining from rigorously specifying such dependencies quickly leads to inconsistencies of the architectural description.
- Architects compose structural models to obtain a unified perspective on (parts of) the architecture, to understand the interactions between elements from different models, and to perform various types of analysis. However, stakeholders demand guarantees that a composed model is a correct representation of the models being composed. The lack of precisely defined dependencies among models makes it hard to compose models correctly and provide such guarantees.
- Tools support for describing dependencies between structural models is indispensably in practice. In particular, automating model composition would significantly increase the design comfort, allowing the architect to quickly obtain a unified perspective on the

Fig. 1. Overview of concepts used in this paper.

architecture and reveal conflicts between models. However, ambiguous descriptions of dependencies between models hampers the development of tool support.

In this paper, we put forward a formally founded approach to manage and exploit dependencies between structural models. The formal specification paves the way to rigorous specification of dependencies between structural models and automation of model composition. Our particular focus is on the composition of structural models and crucial properties of this composition. The approach is based on two key concepts: *relation* and *composition operator*. A relation defines a particular type of overlap between structural models. We formally specify three common relations: unification, submodel, and subelement. A composition operator defines the integration of several structural models based on the relations between these models, yielding a composed structural model. We define well-formedness rules for structural models and relations, and we formally capture properties of model composition, namely traceability and information preservation. We use a formal language based on set theory that provides an accessible notation and compact specification of the core concepts. A complete formal description of the relations and the composition operator in Haskell, including proofs of the properties of model composition is available in [5].

Building upon earlier work [7], we integrated the relations and the composition operator in ArchStudio [6], providing a Java implementation of the relations and composition in the xADL language [11]. As an illustrative case, we use excerpts from a distributed video-on-demand system that has been designed with the tool.

In summary, the key contributions of this paper are: (1) a formal definition of three common *relations* and a *composition operator* for structural models (section 2); (2) a formal specification of well-formedness rules (section 2) and a semantic specification of model composition (section 3); (3) a demonstration tool build in ArchStudio that supports the relations and automated composition in the xADL language (section 4). Section 5 discusses related work. Finally, section 6 discusses experiences and conclusions.

2 A formal model for relations and composition

Before going into the formal details, we introduce the main concepts used in this paper. Fig. 1 shows the concepts in the context of the conceptual model of ISO 42010. [1] The subset of concepts of ISO 42010 is shown in gray.

An *architectural description* (AD) is a collection of products to document a specific architecture. An AD is organized into one or more *architectural views*, where a view is a rep-

[1] ISO 42010 is based on ANSI/IEEE-Std-1471 and is currently undergoing standardization as the standard for “Systems and Software Engineering - Architectural Description”.

resentation of a whole system from the perspective of a related set of concerns. A view is composed of one or more architectural models. A model contains a concrete description of architectural elements. This paper provides an in depth study of structural views. A *structural view* is a base view to describe software systems and is an aggregation of one or more structural models and relations. A *structural model* contains concrete architectural elements like components, connectors and interfaces. A *structural relation* describes a relation between two structural models. Unification, submodel, and subelement are specialization's of this relation. A *composition* integrates several structural models using the relations among these models, defining an integrated model.

The formalization is build up in steps: structural models in 2.1, relations in 2.2 and composition in 2.3. In each step we briefly introduce the specific concepts, we explain the objective of the formalization, we formalize the concepts, and we define well-formedness rules.

2.1 Structural models

We consider the following basic elements:

Structural model A structural model defines components and connectors and how they are linked together. A structural model has a unique name, a set of components and connectors, and a set of links.

Components Components are the loci of computation in the architecture. A component has a unique name, a set of interfaces and an optional substructure.

Connectors Connectors are the loci of communication in the architecture, specified in the same way as components. Connectors can only be used between components.

Interfaces Interfaces are components' and connectors' portals to the outside world. An interface has a unique identifier and a direction. The direction indicates the direction of the information flow, and can be in, out or inout.

Links A link connects two interfaces. Links define the topology of the architecture.

Substructure A substructure defines the internal structures of either a component or a connector. A substructure consists of components, connectors and links, and a mapping of internal elements to the enclosing element with interface mappings.

Interface mapping Interface mappings are used within the context of substructures. An interface mapping defines a mapping between an outer interface (to an interface of an enclosing element) and an inner interface (to an internal element).

Names of elements are fully qualified. For example, an interface `stream` in component `User` that is situated in model `VOD` has as name `VOD.User.stream`. When no confusion is possible, only the last part of the name is used to refer to the element, i.e. `stream`.

We use the architecture of a simplified Video on Demand (VOD) system as a running example. A VOD system allow users to search and watch videos at their demand. The extract used in this paper includes pay-per-view support, where a user pays for the videos she/he wants to see. The left hand side of Fig. 2 shows the VOD structural model, defining the basic service to search and watch videos. The `User` component is responsible for interaction with the user. `Streamer` pushes streams to users based on commands it receives from `Query`. `Query` handles searches and requests from `User`, and`DB` manages the database of videos. Finally, the connector `SC` links `User` with `Streamer`, and `VQC` links `User` with `Query`.

Formalization.

Objective: providing a rigorous description of structural models (1) to avoid ambiguity and inconsistency in descriptions of models; (2) to support the definition of relations and composition.

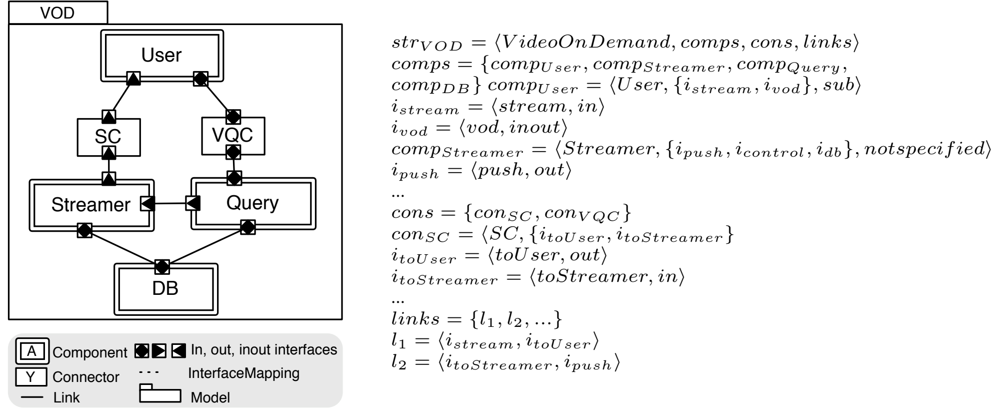


Fig. 2. Example structural model. Left: visual notation. Right: extract of formal notation

The following sets define structural models[2]:

ID	Set of all possible fully qualified names
$DIR = \{in, out, inout\}$	Set of possible directions
$INT \subset ID \times DIR$	Set of interface
$CC \subset ID \times \mathbb{P}\, INT \times SUB$	Set including both components and connectors
$COMP, CON \subset CC$	Set of components and connectors respectively
$LINK \subset INT \times INT$	Set of all Links
$IM \subset INT \times INT$	Set of all interfacemapping
$SUB \subset \mathbb{P}\, COMP \times \mathbb{P}\, CON \times \mathbb{P}\, LINK \times \mathbb{P}\, IM$	Set of all substructures
$notspecified = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \in SUB$	Short notation if no substructure is specified
$STRUCT \subset ID \times \mathbb{P}\, COMP \times \mathbb{P}\, CON \times \mathbb{P}\, LINK$	Set of all structural models

Since several well-formedness rules apply (see below), we use the subset construct to define several of the sets. We refer to an instance of an architectural element using its type name with its name as subscript, e.g. a component A is noted as $\texttt{comp}_A$. Furthermore, we use an abbreviate notation to refer to the internals of an element. For example, to refer to the interfaces of a component $\texttt{comp}_A$ we use the notation $\texttt{comp}_A\texttt{.ints}$. The following conventions on type names and internal parts are used:

$i_{name} = \langle name, dir \rangle$	Interface
$cc_{name} = \langle name, ints, sub \rangle$	CC
$comp_{name}, con_{name} = \langle name, ints, sub \rangle$	Component and connector
$l_i = \langle int1, int2 \rangle$	Link
$sub = \langle comps, cons, links, ims \rangle$	Sub structure
$im_i = \langle inner, outer \rangle$	Interface mapping
$str_{name} = \langle name, comps, cons, links \rangle$	Structural model

As an example, the left hand side of Fig. 2 shows an extract of the formalization of the structural model shown on the right hand side. The formalization specifies the VOD model, the User and Streamer component with interfaces, the SC connector with interfaces, and two links that connect the two components through the connector.

[2] In this paper, we do not explicitly consider the difference between an element and a reference to an element. The extended version of the formalism [5] explicitly considers references.

We define several functions. For the sake of simplicity, we define a function with the same name for several types (separated by a comma). This can be be translated to several functions for each of the types. [5] provides the full specification of these functions.

$ccs : STRUCT, SUB \rightarrow \mathbb{P}\, CC$: returns all components and connectors in the given structure or substructure; we use the abbreviate notation $struc.ccs$ and $sub.ccs$.

$allCC : CC, STRUCT, \mathbb{P}CC, \mathbb{P}STRUCT \rightarrow \mathbb{P}\, CC$: returns recursively all components and connectors in the given element (i.e. all elements in substructures are included).

$ints : CC, STRUCT, \mathbb{P}CC, \mathbb{P}STRUCT \rightarrow \mathbb{P}\, INT$: returns all interfaces that are in the given element (non recursively, so interfaces in substructures are not included).

$allInts : CC, STRUCT, \mathbb{P}CC, \mathbb{P}STRUCT \rightarrow \mathbb{P}\, INT$: returns recursively all interfaces in the given element.

$connected : INT \times INT \rightarrow Bool$: checks if two given interfaces are connected either by a link or by a link and several interface mappings.

Well-formedness rules. The following rules exclude invalid and redundant tuples from the definition of structural models:

$$\begin{gathered} \forall i_1, i_2 \in INT : i_1.name = i_2.name \Rightarrow i_1 = i_2 \\ \forall c_1, c_2 \in CC : c_1.name = c_2.name \Rightarrow c_1 = c_2 \end{gathered} \tag{1}$$

$$\begin{gathered} \forall im \in IM : im.outer \neq im.inner \wedge im.outer.dir = im.inner.dir \\ \forall l \in LINK : (l.int1.dir = inout \wedge l.int2.dir = inout) \vee \\ (l.int1.dir = in \wedge l.int2.dir = out) \vee (l.int1.dir = out \wedge l.int2.dir = in) \end{gathered} \tag{2}$$

$$\begin{gathered} \forall str \in STRUCT : \forall l \in str.links : \{l.int1, l.int2\} \subseteq (ints\ str.comp \cup ints\ str.con) \\ \forall sub \in SUB, \forall l \in sub.links : \{l.int1, l.int2\} \subseteq (ints\ sub.comp \cup ints\ sub.con) \end{gathered} \tag{3}$$

$$\begin{gathered} \forall sub \in SUB, \forall cc \in CC : cc.sub = sub \Rightarrow \\ (\forall im \in sub.im : im.inner \in (allInts\ sub.comp \cup allInts\ sub.con) \wedge im.outer \in cc.ints) \end{gathered} \tag{4}$$

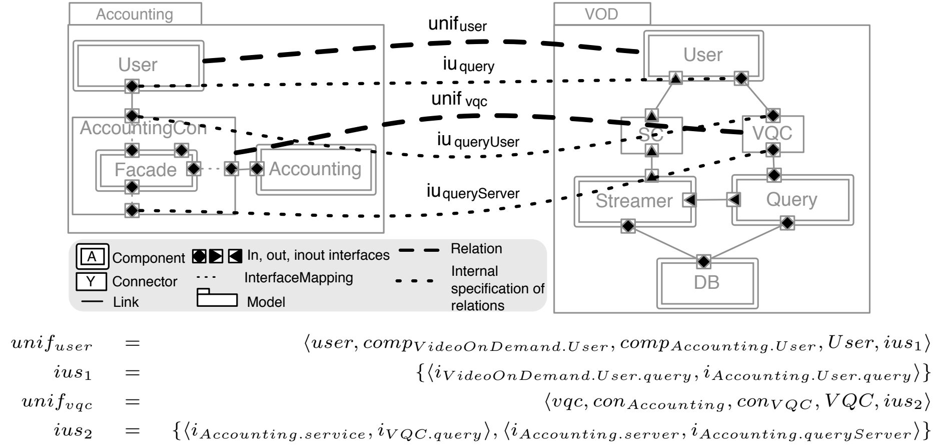
$$\begin{gathered} \forall struct \in STRUCT, \forall con \in struct.con, \exists l \in struct.links : \\ (l.int1 \in con.ints \Rightarrow \exists comp \in struct.comp : l.int2 \in comp.ints) \wedge \\ (l.int2 \in con.ints \Rightarrow \exists comp \in struct.comp : l.int1 \in comp.ints) \end{gathered} \tag{5}$$

Rule 1 ensures that each element with a specific name is unique. The rule is only shown for interfaces and components. For connectors and structures, the definition is very similar. Rule 2 ensures that the directions of interfaces in interface mappings and links are compatible. Rule 3 and rule 4 make sure that links and interface mappings refer to interfaces of the correct structure or substructure. This rule excludes links between interfaces in different models and links between an interface in a substructure and an interface that is not in this substructure. Rule 5 makes sure that connectors can only be used between components. We only show the rule for structural models, the definition for substructures is very similar.

2.2 Structural Relations

Structural relations document dependencies between different structural models. Explicitly specified relations allow an architect to see the implications of changing one model on another model. Relations form the basis for model composition what is subject of the next section. From our experience in several case studies, we deduced the following relations:

- **Unification**: expresses that two elements (either components or connectors) that appear in the two different structural models are the same element. A unification is always companioned with interface unifications that express which interfaces are the same.



$$
\begin{aligned}
unif_{user} &= \langle user, comp_{VideoOnDemand.User}, comp_{Accounting.User}, User, ius_1\rangle \\
ius_1 &= \{\langle i_{VideoOnDemand.User.query}, i_{Accounting.User.query}\rangle\} \\
unif_{vqc} &= \langle vqc, con_{Accounting}, con_{VQC}, VQC, ius_2\rangle \\
ius_2 &= \{\langle i_{Accounting.service}, i_{VQC.query}\rangle, \langle i_{Accounting.server}, i_{Accounting.queryServer}\rangle\}
\end{aligned}
$$

Fig. 3. Example of unification relation. Top: two models with their relation. Bottom: corresponding formal specification of the relations.

- **Submodel**: expresses that a structural model describes the internal structure of an architectural element of another structural model. Interface mappings are used to map internal elements on the enclosing element.
- **Subelements**: expresses that individual elements or groups of elements from one structural model are part of the substructure of an element of another structural model. Similar to submodel, the subelements relation uses interfaces mappings.

Figure 3 illustrates the relations. The left hand side shows the structural model `Accounting`. This model shows a connector that internally uses `Facade` to enforce that users pay to view a video. The right hand side shows the `VOD` model (as in Fig. 2). We visually illustrate the relations between elements with lines going from the one structure to the other structure. In the example, the bold dotted lines represent unifications, the light dotted lines unifications of interfaces. $\texttt{unif}_{User}$ expresses that `Accounting.User` and `VOD.User` are the same component. Even more, one of the interfaces of User is the same as expressed by $\texttt{iu}_{query}$. $\texttt{unif}_{VQC}$ expresses that `Accounting.AccountingCon` and `VOD.VQC` are the same connector, with two interface unifications $\texttt{iu}_{queryUser}$ and $\texttt{iu}_{queryServer}$.

Formalization.

Objective: providing a rigorous description of the three relations (1) to avoid ambiguity and inconsistency in documenting dependencies between structural models; and (2) to support the definition of composition.

The following sets define the relations:

$IU \subset INT \times INT$	Set of all interface unifications
$UNIF \subset ID \times CC \times CC \times ID \times \mathbb{P}\, IU$	Set of all unification relations
$SUBELEM \subset ID \times CC \times \mathbb{P}\, CC \times \mathbb{P}\, IM$	Set of all subelement relations
$SUBMODEL \subset ID \times CC \times STRUCT \times \mathbb{P}\, IM$	Set of all submodel relations
$REL = UNIF \cup SUBELEM \cup SUBMODEL$	The set of all relations.
$iu = \langle ui1, ui2, newName\rangle$	Conventions on names of internals for interface unification
$unif_{name} = \langle name, elem1, elem2, newName, ius\rangle$	Conventions on names of internals for unification
$subelem_{name} = \langle name, target, subelems, ims\rangle$	Conventions on names of internals for subelements
$submodel_{name} = \langle name, target, submodel, ims\rangle$	Conventions on names of internals for submodel

Fig. 3 contains a formal description of the relations in our running example.

Well-formedness rules. The same uniqueness requirement as defined in rule 1 applies for the relations. Furthermore, the function `allCC` is also defined for relations.

$$\forall unif \in UNIF, \exists s_1, s_2 \in STRUCT : (unif.elem1 \in s_1.comps \wedge unif.elem2 \in s_2.comps) \vee (unif.elem1 \in s_1.cons \wedge unif.elem2 \in s_2.cons) \tag{6}$$

$$\forall unif \in UNIF, \forall s_1, s_2 \in STRUCT : unif.elem1 \in s_1.ccs \wedge unif.elem2 \in s_2.ccs \Rightarrow s_1 \neq s_2 \tag{7}$$

$$\forall unif \in UNIF, \forall iu \in unif.ius : (iu.ui1 \in unif.elem1.ints \wedge iu.ui2 \in unif.elem2.ints) \tag{8}$$

$$\forall sub \in SUBELEM, \forall cc \in sub.subelems, \exists s_1, s_2 \in STRUCT : sub.target \in s_1.ccs \wedge cc \in s_2.ccs \wedge s_1 \neq s_2 \tag{9}$$

$$\forall submodel \in SUBMODEL : submodel.target \notin (allCC\ submodel.submodel) \tag{10}$$

$$\forall submodel \in SUBMODEL, \forall im \in submodel.ims, \exists cc \in submodel.submodel.ccs : im.outer \in submodel.target.ints \wedge im.inner \in cc.ints \tag{11}$$

Rule 6 ensures that unifications between a component and a connector are not possible. Rule 7 ensures that two unified elements are always from a two different structural models. Rule 8 ensures that all interface unifications are between the interfaces of the unified elements. Rule 9 ensures the target element and the subelements in a subelements relation are from different models. Rule 10 ensures that the target element is not an element of the submodel. Rule 11 ensures that interface mappings in relations are indeed between the related elements.

2.3 Composition

Composition defines the integration of several structural models based on the relations defined between these models. We distinguish between the *composition specification* and the *composition operator*. The composition specification *selects* models and relations. The composition operator takes a specification and *defines* an integrated structural model and a set of *traces*. The set of traces relate all elements in the output model of a composition operator with the elements in the input models selected by the composition specification.

Formalization.
Objective: providing a rigorous description of model composition and information preservation properties of the composition (1) to provide guarantees to the stakeholders that a composed model is a correct representation of the input models, (2) to support various types of analysis, and (3) to pave the way to automate model composition.
We start with a formal model of the composition specification:

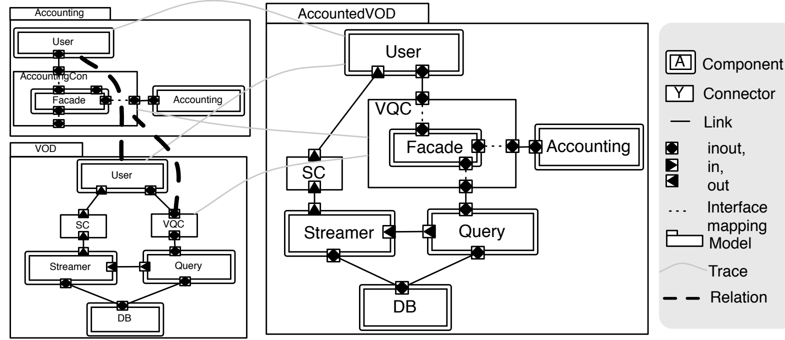
$COMPSPEC \subset ID \times \mathbb{P}\, STRUCT \times \mathbb{P}\, REL$	Set of all composition specifications
$spec_{name} = \langle name, inStr, inRels \rangle$	Conventions on names of internal elements

We use the terms 'input models' (`inStr`) and 'input relations' (`inRels`) to refer to the models and relations selected in the composition specification. Fig. 4 shows an example of a composition specification based on the models and unification relations defined in Fig. 3.

Several well-formedness rules apply:

$$\forall spec \in COMPSPEC, \forall rel_1, rel_2 \in spec.inRels : allCC\ rel_1 \cap allCC\ rel_2 = \emptyset \tag{12}$$

$$\forall spec \in COMPSPEC, \forall rel \in spec.inputRels : allCC\ rel \subseteq allCC\ spec.inputModels \wedge allInts\ rel \subseteq allInts\ spec.inputModels \tag{13}$$



$$spec_{AccountedVOD} = \langle AccountedVOD, \{str_{VOD}, str_{Accounting}\}, \{unif_{user}, unif_{vqc}\}\rangle$$

Fig. 4. Example of composition. Left: the two input models, related by two unifications. Right: the output model. Gray lines from left to right show example traces for the unified elements.

Rule 12 ensures there is no overlap between the input relations. In case of an overlap, it is not always clear what the combined result would look like. For example, one cannot describe in the same composition specification that a component is unified and at the same time is a sub element of another component. Currently, an architect first has to compose the models using a first relation, and afterwards compose the result with another model using the second relation. This is a current limitation of our approach. In some cases these constraints can be relaxed, but additional well-formedness rules are required to make sure that the relations are consistent with each other. Rule 13 ensures that all relations are within the scope of the input models.

Next, we define a composition operator. The composition operator takes a composition specification and defines an integrated structural model and a set of traces. This leads to a function with the following signature:

$$compositionFunction : COMPSPEC \rightarrow STRUCT \times \mathbb{P}\, TRACE$$

We use the term 'output model' (`outStr`) to refer to the model defined by the composition; `traces` denotes the resulting set of traces. Note that the composition function already implies that the output model is well-formed since it is an element of STRUCT for which the well-formedness rules apply. Each trace contains two elements, an element in the output model and an element in the input model. Traces allow us to keep an overview of the interconnections between the input models and the output model. Traces are defined as follows:

$$TRACES = (STRUCT \times STRUCT) \cup (CC \times CC) \cup (INT \times INT)$$
$$trace = \langle out, in\rangle$$

The operational description of the composition operator is too complex to show in this paper; [5] provides all the details. Here, we briefly explain the three main steps of model composition and illustrate the result using the running example. In section 3, we define the semantics of the composition operator by formally capturing information preserving properties.

The left hand side of Fig. 4 shows the two input models (`Accounting` and `VOD`), the right hand side shows the resulting output model (`AccountedVOD`). The thin gray lines from left to right represent the traces for components and connectors (to simplify the figure, we left out the traces for interfaces). The main steps of model composition are:

1. Apply the relations one by one. The order is not relevant since relations are independent (implied by rule 12). Applying means: (i) use the information in the relation to

define an integrated architectural element and add this element in the output model, and (ii) add traces between the respective elements in the input models and the output model. For example, applying the relation $\texttt{unif}_{User}$ yields: (i) the `User` component with two interfaces in the output model, and (ii) two traces for the `User` component from the output model to two `User` components in the input models, and a set of traces for the interfaces. Similarly, relation $\texttt{unif}_{VQC}$ adds the `VQC` connector and its interfaces, and a set of traces for all elements. Notice that the internal structure of `Accounting.AccountingCon` is preserved by the unification.
2. For each component and connector that can not be found in the traces: copy the element from the input models to the output model, and add the respective traces for the element. For our running example, this step adds the elements `Accounting`, `Streamer`, `Query`, `DB`, `SC` to the output model, and traces for all the elements and their internals.
3. Link the components and connectors in the output model. Each link in the input models is translated to a link in the output model using the traces.

3 Information preservation properties of the composition operator

The formal specification of model composition allows us to formally capture information preservation properties of the composition operator. Information preservation properties provide guarantees to the stakeholders that the output model is a correct representation of the composed models. We discuss four properties: traceability, consistency, model completeness, and relational completeness. For each property we give a brief introduction, we explain the objective of the formalization, and we formally define the property. Proofs of the properties can be found in [5].

3.1 Traceability

Traceability has widely been recognized as an important aspect of software development. IEEE [1] defines traceability as "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship; for example, the degree to which the requirements and design of a given software component match." The composition operator presented in this paper supports traceability between the input models and the output model of a composition.

Formalization.

Objective: ensuring that all elements from the output model are traceable to elements from the input models (1) to enable architects to trace back the origin of each element of a composed model; (2) to provide the basis for describing and proving the other information preserving properties, and (3) to pave the way for automatic analysis, such as consistency checking and impact analysis.

The formal definition of traceability is captured as folows:

$$\forall str_i \in inStrs : \langle outStr, str_i \rangle \in traces \tag{14}$$

$$\forall c_{out} \in allCC\ outStr, \exists str \in inStrs, \exists c_{in} \in allCC\ str : \langle c_{out}, c_{in} \rangle \in traces \tag{15}$$

$$\begin{aligned}&\forall c_{out} \in (allCC\ outStr), \forall i_{out} \in c_{out}.ints, \exists str \in inStrs, \exists c_{in} \in (allCC\ str), \exists i_{in} \in c_{in}.ints : \\ &\langle c_{out}, c_{in} \rangle \in traces \wedge \langle i_{out}, i_{in} \rangle \in traces\end{aligned} \tag{16}$$

$$\begin{gathered}\forall i_{out1}, i_{out2} \in (allInts\ outStr), \forall str \in inStrs, \exists i_{in1}, i_{in2} \in (allInts\ str) : \\ (\langle i_{out1}, i_{in1}\rangle \in traces \wedge \langle i_{out2}, i_{in2}\rangle \in traces\ \wedge connected\ i_{out1}\ i_{out2}) \Rightarrow connected\ i_{in1}\ i_{in2}\end{gathered} \tag{17}$$

Equation 14 states that there is a trace between the output model and each structural input model of a model composition. Equation 15 states that for each component and connector in the output model, there exists a corresponding component or connector in the input models, identifiable through traces. Equation 16 states that for each interface of each component and connector in the output model, there exists an interface of the corresponding component or connector in the input models, identifiable through traces. One could interpret traces as a 'binary surjective relation' between the set of all elements with a name in the output model, and the set of all elements with a name in the input models. Equation 17 states that each connection in the output model can be traced back to a connection in the input model. Two interfaces are connected if there is a link between the interfaces, or between interface mappings of the interfaces (captured in the connected function defined in section 2.1). Note that links are not directly traceable. Traces of links are implied by the connections.

3.2 Consistency

Consistency refers to the compatibility of the output model with the input structural model. In particular, the output model of a model composition should not contradict the input models. We define two consistency properties: (1) two separate components of an input model cannot become a single component in the output model, and (2) if an input model defines that an element is part of a substructure, it must be part of the same substructure in the output model.

Formalization.

Objective: ensuring that the output model is consistent with the input models (1) to make the guarantees of consistency of model composition explicit; (2) to enable automation of consistency checks.

Model composition must preserve the distinction between elements defined in the input models. For example, if an input model contains components A and B, it is not allowed that the output model contains a single component that represents both A and B. This would imply that the composition does not respect the distinction between components A and B in the input model. The same holds for interfaces. This is captured in the following two formal statements:

$$\begin{gathered}\forall c_{out} \in allCC\ outStr, \forall str_1, str_2 \in inStrs, \exists c_1 \in allCC\ str_1, \exists c_2 \in allCC\ str_2 : \\ (\langle c_{out}, c_1\rangle \in traces \wedge \langle c_{out}, c_2\rangle \in traces \wedge c_1 \neq c_2) \Rightarrow str_1 \neq str_2\end{gathered} \tag{18}$$

$$\begin{gathered}\forall c_{out} \in allCC\ outStr, \forall i_{out} \in c_{out}.ints, \exists str_1, str_2 \in inStrs, \exists c_1 \in (allCC\ str_1), \\ \exists c_2 \in (allCC\ str_2), \exists i_1 \in c_1.ints, \exists i_2 \in c_2.ints : \\ \langle c_{out}, c_1\rangle \in traces \wedge \langle c_{out}, c_2\rangle \in traces \wedge \langle i_{out}, i_1\rangle \in traces \wedge \langle i_{out}, i_2\rangle \in traces \wedge i_1 \neq i_2 \\ \Downarrow \\ str_1 \neq str_2\end{gathered} \tag{19}$$

Equation 18 states that for each component and connector in an output model, if there are two corresponding components or connectors in the input models, these components or connectors must be from different input models. Equation 19 states that for each interface of a component or connector in the output model, if there are two corresponding interfaces in the input models, these interfaces must be from different input models.

Next to preservation of the differences between elements, substructures must be preserved. This is captured in the following equations:

$$\forall c_{in} \in inStrs.ccs, \forall c_{insub} \in c_{in}, \nexists c_{out} \in outStr.ccs : \langle c_{out}, c_{in} \rangle \in traces \tag{20}$$

$$\begin{gathered}\forall c_{in} \in inStrs.ccs, \forall c_{insub} \in c_{in}, \exists c_{out} \in outStr.ccs, \exists c_{outsub} \in c_{out}.sub.ccs : \\ \langle c_{out}, c_{in} \rangle \in traces) \Rightarrow \langle c_{out}, c_{in} \rangle \in traces\end{gathered} \tag{21}$$

Equation 20 states that if an element is part of a substructure in the input models, there exists no corresponding element that is not part of a substructure in the output model. Equation 21 states that if an element is part of a substructure of an input element c_{in}, and there exists a corresponding element part of a substructure of output element c_{out}, there must be a trace between c_{in} and c_{out}. This makes sure that the element is part of the same substructure in the output model.

3.3 Model completeness

Model completeness refers to the inclusion of all information of the input models in the output model of a model composition.

Formalization.

Objective: ensuring that the output model of a model composition covers all information covered in the input models (1) to make the guarantees of information preservation of input models explicit; (2) to enable automation of completeness checks.

Model completeness requires that for each element in the input models there must be a corresponding element in the output model, identifiable through traces. Together with the consistency property (section 3.2) this implies that the output model covers all information of the input models. Model completeness is captured in the following formal statements:

$$\forall str_{in} \in inStrs, \forall c_{in} \in allCC\ str_{in}, \exists! c_{out} \in allCC\ outStr : \langle c_{out}, c_{in} \rangle \in traces \tag{22}$$

$$\begin{gathered}\forall str \in inStrs, \forall c_{in} \in allCC\ str, \forall i_{in} \in ccInts\ c_{in}, \exists! c_{out} \in allCC\ outStr, \\ \exists! i_{out} \in ccInts\ c_{out} : \langle c_{out}, c_{in} \rangle \in traces \wedge \langle i_{out}, i_{in} \rangle \in traces\end{gathered} \tag{23}$$

$$\begin{gathered}\forall str \in inStrs, \forall c_{in} \in (allCC\ str), \exists! c_{out} \in (allCC\ outStr) : \\ \langle c_{out}, c_{in} \rangle \in traces \wedge (c_{in}.sub = notspecified \vee \\ (\forall c_{insub} \in allCC\ c_{in} : \exists! c_{outsub} \in allCC\ c_{out} : \langle c_{outsub}, c_{insub} \rangle \in traces))\end{gathered} \tag{24}$$

$$\begin{gathered}\forall str \in inStrs, \forall i_{in1}, i_{in2} \in (allInts\ str), \exists! i_{out1}, i_{out2} \in (allInts\ outStr) : \\ \langle i_{out1}, i_{in1} \rangle \in traces \wedge \langle i_{out2}, i_{in2} \rangle \in traces \wedge connected\ i_{in1}\ i_{in2} \\ \Downarrow \\ connected\ i_{out1}\ i_{out2}\end{gathered} \tag{25}$$

Equation 22 states that for each component and connector in the input models, there exist a unique counterpart in the output model, identifiable through traces. Equation 23 states that for each interface of each component and connector in the input models, there exists a corresponding interface in the uniquely corresponding component or connector of the output model. Equation 24 states that for each component and connector in the input models, the corresponding component or connectors in the output model has a corresponding substructure. Equation 25 states that for each connection between interfaces in the input models, there exists a connection between the corresponding interfaces in the output model. Notice that the completeness property implies that the set of traces can be interpreted as a total surjective function between all elements in the input models with a name and all elements in the output model with a name.

3.4 Relational completeness

Relational completeness refers to the inclusion of all information of the relations between the input models in the output model of a model composition.

Formalization.

Objective: ensuring that the output model of a model composition covers all information covered in the relations between the input models (1) to make the guarantees of information preservation of relations between input models explicit; (2) to enable automation of completeness checks.

The following equations formalize relational completeness. The equations have to be considered in the context of the already defined properties:

$$\begin{gathered}\forall unif_i \in inRel \cap UNIF, \exists! cc_{out} \in (allCC\ outStr), \exists t_1, t_2 \in traces : \\ \begin{pmatrix} t_1 = \langle cc_{out}, unif_i.elem1 \rangle \wedge t_2 = \langle cc_{out}, unif_i.elem2 \rangle \wedge \\ (\nexists t_3 \in trace : t_3.out = cc_{out} \wedge t_3 \neq t_1 \wedge t_3 \neq t_2) \end{pmatrix} \wedge \\ \begin{pmatrix} \forall iu_j \in unif_i.ius, \exists! i_{out} \in cc_{out}.ints, \exists t_1, t_2 \in traces : \\ t_1 = \langle i_{out}, iu_j.ui1 \rangle \wedge t_2 = \langle i_{out}, iu_j.ui2 \rangle \wedge \\ (\nexists t_3 \in trace : t_3.out = i_{out} \wedge t_3 \neq t_1 \wedge t_3 \neq t_2) \end{pmatrix} \wedge \\ \begin{pmatrix} \forall i_{in} \in (unif_i.elem1.ints \cup unif_i.elem2.ints)/allInts\ unif_i.ius, \\ \exists i_k \in cc_{out}.ints : \langle i_k, i_{in} \rangle \in traces \end{pmatrix}\end{gathered} \tag{26}$$

$$\begin{gathered}\forall submodel_i \in inRel \cap SUBMODEL, \exists! cc_{out} \in (allCC\ outStr) : \\ \langle cc_{out}, submodel_i.target \rangle \in traces \wedge \\ (\forall cc \in submodel_i.submodel.ccs, \exists! cc_{subout} \in cc_{out}.sub : \langle cc_{subout}, cc \rangle \in traces)\end{gathered} \tag{27}$$

$$\begin{gathered}\forall subelem_i \in inRel \cap SUBELEM, \exists! cc_{out} \in (allCC\ outStr) : \\ \langle cc_{out}, subelem_i.target \rangle \in traces \wedge \\ (\forall cc \in subelem_i.subelems, \exists! cc_{subout} \in cc_{out}.sub : \langle cc_{subout}, cc \rangle \in traces)\end{gathered} \tag{28}$$

$$\begin{gathered}\forall str \in inStrs, \forall cc_{in} \in str.ccs/allCC\ inRel, \exists! cc_{out} \in strOut, \exists t_{c1} \in traces : \\ t_{c1} = \langle cc_{out}, cc_{in} \rangle \wedge (\nexists t_{c2} \in traces : t_{c2}.out = cc_{out} \wedge t_{c2} \neq t_{c1}) \wedge \\ \begin{pmatrix} \forall i_{in} \in cc_{in}.ints, \exists! i_{out} \in cc_{out.ints}, t_{i1} \in traces : t_{i1} = \langle i_{in}, i_{out} \rangle \wedge \\ (\nexists t_{i2} \in traces : t_{i2}.out = i_{out} \wedge t_{i2} \neq t_{i1}) \end{pmatrix}\end{gathered} \tag{29}$$

Equation 26 defines what information is preserved from a unification. The first part of the equation (lines 2-3) captures information preservation with respect to components and connectors. The second part (lines 4-6) captures information preservation of unified interfaces. The last part (lines 7-8) captures information preservation for non-unified interfaces. Equation 27 defines what information is preserved for a submodel relation; equation 28 defines information preservation for a subelems relation. Finally, equation 29 states that all components and connectors (and their interfaces) that are not part of a relation in the input moels must be uniquely traceable to an element in one of the output model. This last equation partly overlaps with the definition of completeness, but with the additional requirement that there is a *unique* trace.

4 xADLComposition: a tool for relations and composition in xADL

To make model composition practical, tools support is indispensably. Tool support allows architects to quickly obtain a unified perspective on the architecture and reveal conflicts between models. An unambiguous description of relations and composition is a prerequisite for developing proper tool support. The formalization presented in the previous sections provides such unambiguous description and served as a necessary basis for the development of

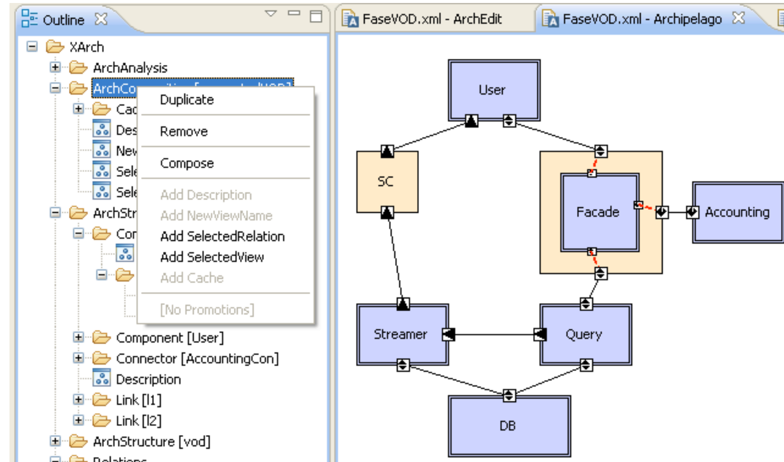


Fig. 5. Snapshot of the tool when the composition of AccountedVOD is triggered (as specified in Fig. 4)

the tools. Moreover, the information preservation properties explicitly describe guarantees about the composition operator.

We integrated the relations and the composition operator in ArchStudio [6], providing a Java implementation of the relations and composition in the xADL language [11]. The extension manifests itself in two aspects: a *language extension* introducing relations and compositon, and an extension of the tool to enable *automatic composition*. We also extended the *visual tool* to show substructures. In the current version of the tool, composition is triggered manually generating the resulting output model and the accompanying set of traces. Fig. 5 shows a snapshot of the tool.

The formal specification of relations and composition provided a sound basis to guide the designers and developers building a consistent and conceptually sound tool. Especially the introduction of traces proved to be a very useful. During composition, the set of traces always contains an up-to-date mapping between the output model and input models of all elements that are already processed by the composition operator. This allows to go back and forth between output and input and check whether elements are already processed during composition.

Several extensions of the tool are planned as future work. (1) We plan to add a visual and intuitive interface to specify relations. (2) Currently, the tool requires explicit triggering to start a composition. We plan to enable automatical synchronization of a composed model if one of the input models changes. (3) Finally, we plan to include support for automatic consistency checks. In particular, we plan to develop tool support to automatically check compatibility of relations between models. More information about the tool can be found on [6]. The website includes several examples of composition, screenshots and demonstration videos, and the source code of the tool.

5 Related work

In previous work we performed a broad study of relations between views and models and proposed a conceptual framework to characterize the relations in terms of usage, scope and mechanism [8]. Here we focus on a number of closely related approaches for composition.

Abi-Antoun et al. [2] describe an algorithm and tool for automatic differencing and merging models, focussing on reconciling different version of the same model. Contrary to our approach relations are not manually specified, but merging is done based on names and structural similarities which are very common for different versions of the same model. Automatic identification of relations based on names could be of interest for our approach too. Architectural stratification [3] relates several architectural strata (structural models in ISO 42010 terminology). Each stratum represents the architectural structure on a certain level of abstraction, and step-wise refinement is used to specify the strata. In each step, a refinement transformation is applied that breaks up a connector in several elements introducing a particular concern. The submodel and subelements relations in our approach are inspired by this work. The approach presented in this paper is not limited to refinement and allows automated integration of models based on the relations between the models. Several authors focus on the integration of several types of models. In [13, 12], Egyed et al. put forward a view integration framework to enable automated design analysis of several heterogeneous view (models in ISO 42010 terminology). The main contribution is on transforming the information from heterogeneous views to allow a comparison, for example to check consistency. Ehrig et al. [14] use views in visual languages and graph morphisms to integrate different views, focussing on extending graph morphisms to easily allow introducing new model types. The focus of the research in this paper is not on heterogeneous model types, but on an in depth study of composition of structural models, including a formal proof of the properties of composition and tool support [5].

As a final remark, there is a relation between the work described in this paper and aspect orientation and model transformation. For aspect orientation, composition (often referred to as weaving) is used to integrate crosscutting concerns with the basis system. Some aspect oriented techniques for architecture (e.g. [16]), detailed design (e.g. Theme/UML [10]), and programming (e.g. HyperJ [18]) use similar ways of composing structures. For model transformation, OMG recently defined a a standard called the Query/View/Transformation Specification (QVT) [17]. QVT advocates specifying relations explicitly and use the relations in transformation definitions. Our approach can be considered as one concrete instantiation of the QVT model, where a transformation corresponds to a composition of structural models.

6 Discussion and conclusion

Managing the dependencies between structural models remains a challenging yet crucial task for maintaining consistency of an architectural description. In this paper we have put forward a formally founded approach to manage and exploit such dependencies. We formalized *structural models*, three common *relations* and a *composition operator* for structural models. We defined well-formedness rules for structural models and relations, and we formally captured information preservation properties of model composition, namely traceability, consistency and completeness. As a proof of concept, we have built a demonstration tool in ArchStudio that supports the relations and automated composition in the xADL language.

This paper achieves the following : (1) the formal description of structural models avoids ambiguity and inconsistency of descriptions of *structural models*; (2) the formal description of the three relations avoids ambiguity and inconsistency in documenting *dependencies between* structural models; and (3) the formal description of model composition and information preservation properties of the composition make *explicit* the *guarantees* of the composition operator. Together, the formal description of (1-3) paved the way to *automate composition* in a tool.

The work in this paper fits in a larger project to support relations and composition of multiple views. In this paper, we have focussed on an in depth study of structural views. From our experience, formalization and providing tool support for structural relations and composition, and in particular providing guarantees about model composition, is non-trivial. But, obviously relations also exist between different views and different types of models. Currently, we are broadening our research to different types of models. We already performed first steps towards supporting a deployment relation between a component & connector model and a runtime infrastructure model to generate a deployment view. Initial experiments are promising. Support for relations with statecharts is currently also under development.

References

1. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12*, Dec 1990.
2. M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and merging of architectural views. *Automated Software Engineering.*, 15(1):35–74, 2008.
3. C. Atkinson and T. Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.
4. L. Bass, P. Clements, and R. Kazman. *Software Architectures in Practice*. Addison, 2003.
5. N. Boucké. Formal proofs of well-formedness and information-preservation properties and haskell source code. `http://www.cs.kuleuven.be/~nelis/composition.htm`.
6. N. Boucké. xADLComposition: a tool for view composition in xADL. `http://www.cs.kuleuven.be/~nelis/xADLComposition`.
7. N. Boucké and T. Holvoet. View composition in multi-agent architectures. *Special issue on Multiagent systems and software architecture, International Journal of Agent-Oriented Software Engineering (IJAOSE)*, 2(2):3–33, 2008.
8. N. Boucké, D. Weyns, R. Hilliard, and T. Holvoet. Characterizing relations between architectural views. *Proceedings European Conference on Software Architecture*, LNCS 5292:66–81, 2008.
9. N. Boucké, D. Weyns, K. Schelfthout, and T. Holvoet. Applying the ATAM to an architecture for decentralized control of a transportation system. In *Quality of Software Architectures conference (QoSA)*, volume LNCS 4214, 2006.
10. S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley, 2005.
11. E. Dashofy, A. van der Hoek, and R. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005.
12. A. Egyed. *Heterogeneous view integration and its automation*. PhD thesis, Los Angeles, CA, USA, 2000. Adviser-Barry William Boehm.
13. A. Egyed and N. Medvidovic. A formal approach to heterogeneous software modeling. In *FASE*, pages 178–192, 2000.
14. H. Ehrig, K. Ehrig, C. Ermel, and U. Prange. Consistent integration of models based on views of visual languages. In *Fundamental Approaches to Software Engineering*. LNCS, 2008.
15. A. Helleboogh, D. Weyns, T. Holvoet, and N. Boucké. On adls and tool support for documenting view-based architectural descriptions. In *Fourth SEI Software Architecture Technlogy User Network Workshop, SATURN*, 2008.
16. M. Kandé. *A Concern-Oriented Approach to Software Architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.
17. OMG. *Meta Object Facility 2.0: Query/View/Transformation Specification*, August 2007.
18. P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Int. Conf. on Software Engineering*, pages 107–119, 1999.
19. D. Weyns, T. Holvoet, K. Schelfthout, and J. Wielemans. Decentralized control of automatic guided vehicles: Applying multi-agent systems in practice. In *Development Track OOPLSA, Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2008.