Composition of Architectural Models: Empirical Analysis and Language Support

Nelis Boucké*, Danny Weyns*, Tom Holvoet*

DistriNet,KULeuven
Celestijnenlaan 200A, 3001, Leuven, Belgium
Tel: +3216327825, Fax: +3216327996

Abstract

Managing the architectural description (AD) of a complex software system and maintaining consistency among the different models is a demanding task. To understand the underlying problems, we analyze several non-trivial software architectures. The empirical study shows that a substantial amount of information of ADs is repeated, mainly by integrating information of different models in new models. Closer examination reveals that the absence of rigorously specified dependencies among models and the lack of support for automated composition of models are primary causes of management and consistency problems in software architecture. To tackle these problems, we introduce an approach in which compositions of models, together with relations among models, are explicitly supported in the ADL. We introduce these concepts formally and discuss a proof-of-concept instantiation of composition in xADL and its supporting tools. The approach is evaluated by comparing the original and revised ADs in an empirical study. The study indicates that our approach reduces the number of manually specified elements by 29%, and reduces the number of manual changes to elements for several realistic change scenarios by 52%.

Keywords: composition, relations, software architecture, architectural models, empirical analysis, architectural description language (ADL)

1. Introduction

As the size and complexity of software systems increases, designing the systems becomes more challenging. Software architecture plays a prominent role in keeping software-intensive systems manageable. Bass et al. (2003) state that an *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, together with the environment in which it is deployed (ISO (2007)).

ISO (2007) defines that the documentation of software architecture, i.e. the 'architectural description' (AD), consists of views and models. Every architectural view consists of architectural models to describe the architecture from the perspective of a related set of stakeholder concerns. An individual model describes concrete architectural elements. Examples are component and connector (C&C) models, communicating processes, deployment and work assignment models. The use of views and models allows to divide the architecture in manageable and comprehensible pieces.

An AD typically contains a substantial amount of information that is repeated in several models throughout the AD. In particular, there are models that mainly join information from other models. We refer to these models as *integrated models*. Inte-

*Corresponding author

Email addresses: nelis.boucke@cs.kuleuven.be (Nelis Boucké), danny.weyns@cs.kuleuven.be (Danny Weyns), tom.holvoet@cs.kuleuven.be (Tom Holvoet)

grated models are responsible for a large part of the repetition. Describing them is common for obtaining an overview (Nejati et al. (2007)), for understanding the interactions (Egyed (2000); Giese and Vilbig (2006)), for bringing features together (Jayaraman et al. (2007)) or for communicating with stakeholders (Clements et al. (2003)).

Preserving consistency in the presence of repetition in the models is a non-trivial task. The fact that software is typically in constant evolution makes maintaining a consistent AD even more complex. Since rigorous descriptions of the dependencies between models are often lacking, consistent handling of changes requires a lot of effort. This is especially true for the integrated models.

This paper investigates in three steps how automated composition can help prevent problems associated with repetition in integrated models.

First, we present an empirical study on the ADs of three non-trivial software systems in section 2. This study yields quantitative statements on the problems associated with repetition, in particular with respect to integrated models. The empirical study identifies opportunities for relations and compositions and defines our goals in extending ADs.

Secondly, we introduce an ADL-based composition approach in order to reach these goals (section 3 and 4). The approach consists of an explicit specification of relations and compositions of models in an ADL. We introduce these concepts formally, and provide a proof-of-concept instantiation of composition in xADL and its supporting tools. As a result, an architect no longer specifies the integrated models directly. Instead the architect specifies a composition, which leads to the same integrated model.

Next, we evaluate our approach in section 6. The evaluation empirically compares the original and revised ADs of the software systems included in section 2.

Finally, the remaining two sections discuss related work and conclusions.

2. Empirical Study and Goals

To investigate the problems associated with integrated models, we performed an empirical study on several existing ADs. The objective of this section is to discuss the problems with integrated models that we encountered during this empirical study (2.1) and to identify the high-level goals for a composition approach (2.2). We come back on the empirical study in section 6 to evaluate our composition approach.

2.1. Empirical Study

2.1.1. Hypotheses

We formulated three hypotheses. The first hypothesis is that an AD contains a lot of repetition (e.g. elements repeated in multiple models) (H1). The second hypothesis is that a large part of this repetition can be attributed to integrated models (H2). The third hypothesis states that for each change to an AD, a large part of the elements needing changes are in integrated models, and thus are repeated updates (H3). These hypotheses are defined with the following assumption in mind: more (manual) repetition in an architectural description is a bad thing, because (1) it requires more effort to specify, (2) changes in combination with repetition are cumbersome and often a source of inconsistencies.

2.1.2. Empirical Procedure

The input of the empirical study consists of three existing ADs for non-trivial systems. The three ADs were specified independent of this study. The first system is an automatic guided vehicle transportation system (AGVTS), a fully automated system that uses unmanned vehicles to provide logistic services in an industrial environment such as a warehouse or a factory. Weyns and Holvoet (2008) specifies more details on this AGVTS. The second system is a digital publishing system (DPS), a next-generation end-to-end media platform using various wired and wireless communication channels for publishing, allowing personalized services based on user-profile and context. Van Landuyt et al. (2006) and Landuyt et al. (2008) specify more details on the DPS. The third system is a modern transport management system (TMS) to cope with ever-changing road circumstances, such as road works, traffic diversions and emergency clearances.

These three example systems are real, mid-size complex distributed systems, with an AD containing multiple views and models. The respective sizes

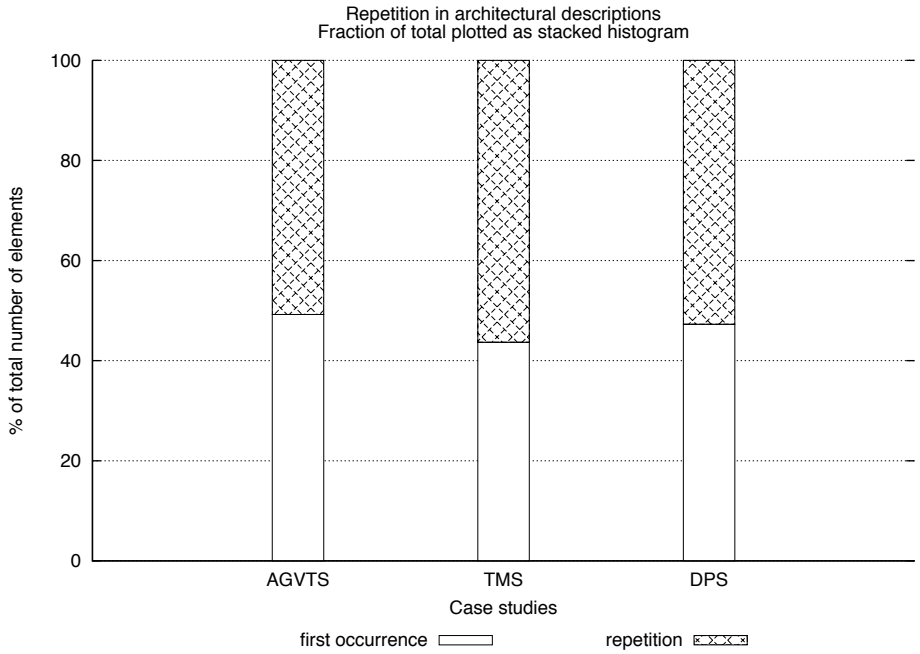


Figure 1: Analysis of repetition: Repetition in ADs.

of the ADs are: 6 models and 130 architectural elements (such as component, interface and link) for AGVTS, 7 models and 341 elements for the TMS and 8 models and 683 elements for the DPS. The details of these cases are not crucial for this paper. We only discuss the TMS in more details below to illustrate the concepts of this paper.

From the original ADs we only consider the C&C, infrastructure and deployment models, as they make up the largest part of the ADs. To ensure a fair comparison, we align the notation of the three original ADs with each other (according to the basic ADL introduced further in section 3.2). This alignment is fairly simple and preserves the original semantics and specification in the models (e.g. from a custom UML like notation to our basic ADL).

To the best of our knowledge, there are no existing metrics to quantify repetition and changes across architectural models. Related empirical studies by Oliveira et al. (2008) and Chitchyan et al. (2009) propose metrics on the architectural level to quantify conflict rates, scaffolding, stability of compositions and expressiveness of compositions, but they cannot be used to confirm or refute our hypotheses. To avoid biassing the metrics, we aim to keep the metrics simple and self-explaining based on counting of elements (further discussed in the section on threads to validity). Measuring is done by manual counting of elements in the different models.

We successively discuss the hypotheses, make an observation on consistency and couple the results to the motivating problems of our research.

2.1.3. Hypothesis 1&2: Repetition and Integrated Models

Elements can be repeated in multiple models. Figure 1 shows repetition as a fraction of the total amount of elements for the three case studies. To calculate repetition, we distinguish between a first occurrence of an element in a model (white region), and all other occurrences in other models which are considered repetitions (shaded region). Possible elements are components, connectors, interfaces, links, interface mappings, nodes or communication channels. Details on the types of elements can be found in section 3.2. Surprisingly, more than half of the ADs are repetitions (between 51% and 56%). This confirms the first hypothesis and indicates that architects spend a lot of effort in repeating architectural elements.

To see what causes repetition we further analyze repetition. This analysis reveals that a large part of repetition is caused by models that integrate infor-

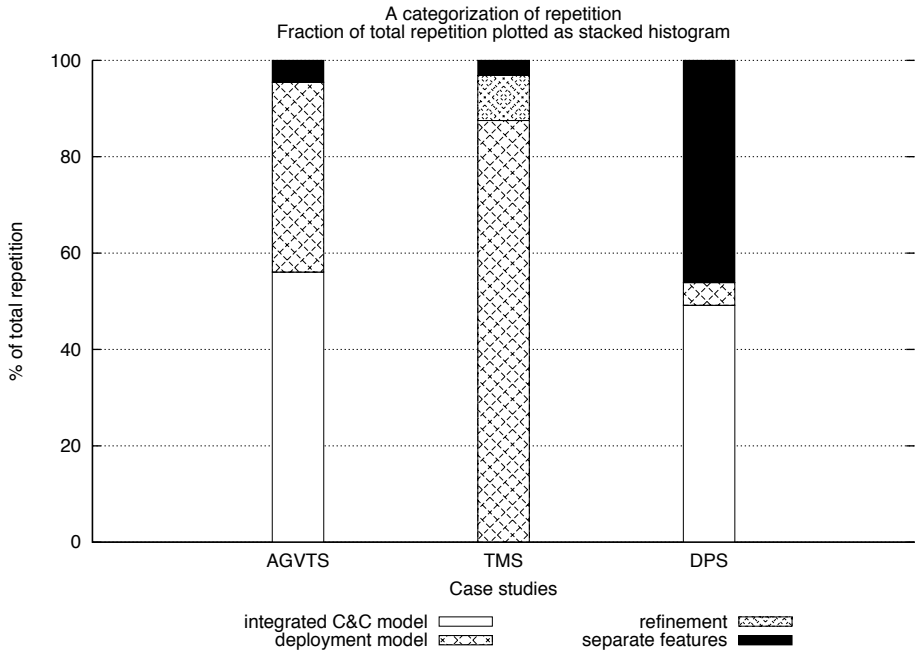


Figure 2: Analysis of repetition: Classes of repetition.

mation from other models (integrated models), including *integrated C&C models* and *deployment models*. For example, several of the architectural descriptions contain an integrated C&C model. Such model shows nothing more than the integrated information from several other models (e.g. to provide an overview) without adding additional information. Figure 3 shows an example of an integrated C&C model in the TMS. The models above the arrow (Overview, Data and Actuator) are integrated with each other in the model below the arrow (Integrated). The details of the models are not important at this stage, but are discussed later.

Repetition can also be caused by refining components from one model in another model (*refinement*) and by models describing *overlapping features*. Each of these causes is considered as a class of repetition. Figure 2 provides an overview of the fraction of repetition in each class with respect to the total repetition.

The results show that integrated C&C models and deployment models are responsible for a large part of the repetition. This confirms the second hypothesis, although it also shows that any solution to cope with integrated models will not take away all repetition. Integrated models are the focus of the remainder of this analysis and chapter.

In all three case studies the architects constructed the integrated models by copying/pasting and manually integrating the information. This indicates that an architect spends a lot of time repeating information on integrated models.

2.1.4. Hypothesis 3: Changes

Keeping the integrated models consistent in the context of evolution is a challenge. We studied the effects of several change scenarios on the TMS architectural description, in particular the integrated models. These scenarios were proposed by the original architects and are representative for real changes during the life cycle of the TMS. The changes to the architecture are: (S1) adding interaction with external media; (S2) adding a new type of actuators; (S3) adding details (a substructure) on data management (distinction between validation and collection); and (S4) adding a deployment alternative.

The nature of the changes to the system is relatively simple. Scenarios S1 and S2 add a component to the system, and connect it through appropriate interfaces, links and connectors to the remainder of the system. Change scenario S3 and S4 also add components and connect them to the remainder of the system, but these scenarios also add additional models to the architectural description. For S3 this is a model

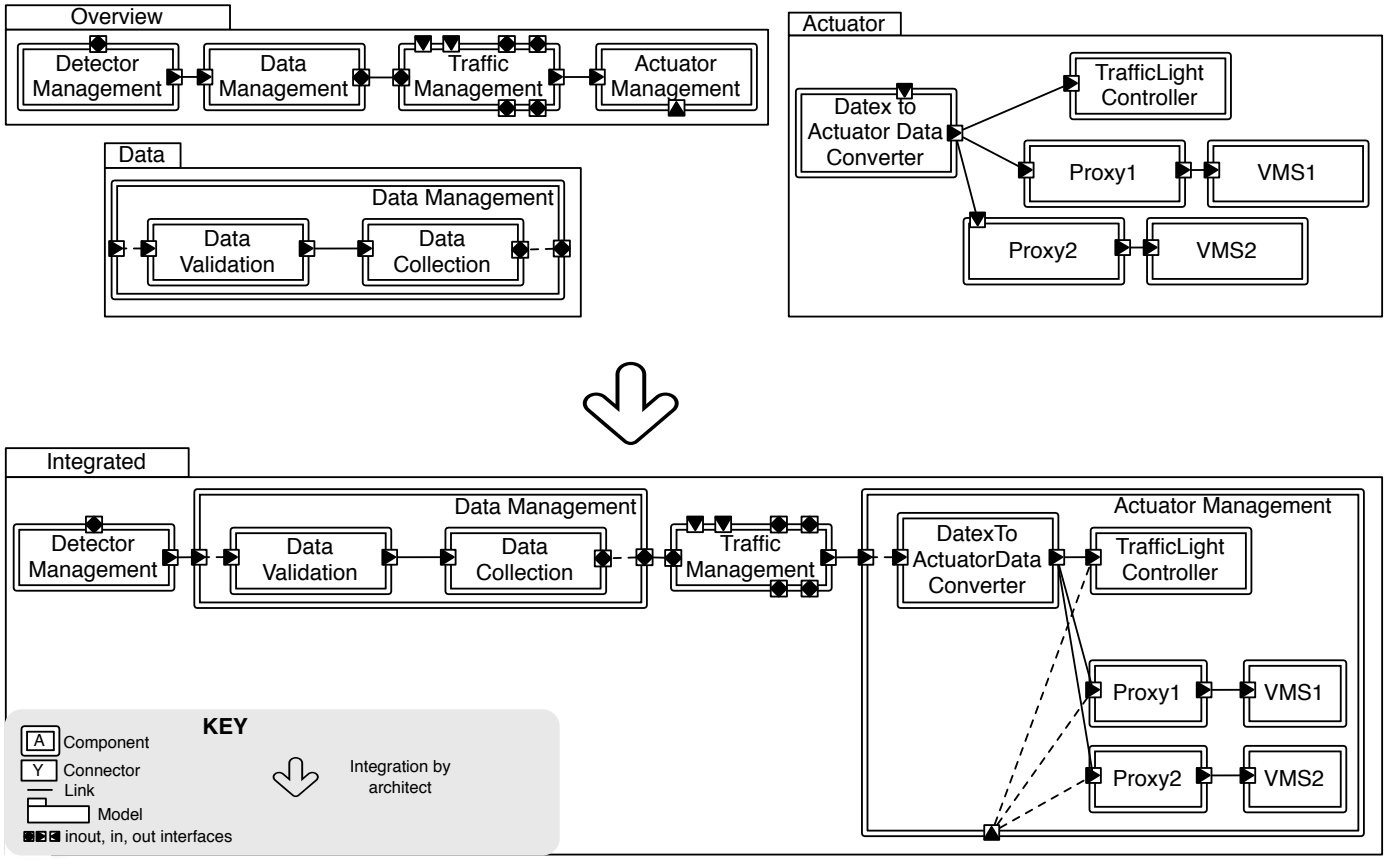


Figure 3: Example of an integrated model: the models above the arrow (Overview, Data and Actuator) are integrated in the model below the arrow (called Integrated).

Table 1: Overview of the number of changes required for each scenario.

	S1	S2	S3	S4
total #changes	21	26	33	103
#changes in integrated	12	18	28	103
% integrated	57%	69%	76%	100%

specifying the substructure of the Data Management component, for S4 this is an infrastructure and deployment model for the new deployment alternative.

The Molesini et al. (2008) metric for measuring architectural change propagation inspired our metrics for measuring changes. The metrics of Molesini do not take multiple models into account but consider the architecture as a monolithic whole, while we distinguish in which model the change is being made. For each of the scenarios, we measured the total number of changes and the changes in integrated models. One change is the addition, removal or alternation of an architectural element. An element is a component, connector, interface, link, interface mapping, node or communication channel. Table 1 contains a summary of the number of changes for each change scenario.

The results show that more than half of the changes take place in integrated models. This confirms the third hypothesis. These changes required manual intervention as there is no mechanism to automatically synchronize integrated models with the models that are integrated.

An observation from our study is that consistent updating of an AD is difficult because there is no rigorous description of the relations between the models. If one model is updated, an architect may need to review all other models to see if they need updating as well.

2.1.5. A Remark on Inconsistencies

One observation during the empirical study is that we found several inconsistencies in each of the ADs. Typically, an inconsistency is an element, such as components, interfaces or links, that is shown in one model but lacking in another while being essential for understanding that part of the design. Other typical inconsistencies are contradictions in terms of links between component. For example, in the TMS we found that a deployment model lacks an essential component and several deployment models lack several links. Most of the inconsistencies we found are between integrated models and the models being integrated. Nearly all integrated models contained one or more inconsistencies.

Note that the lack of rigorous specification of relations between the models hampers the identification of inconsistencies. We found the inconsistencies as a side effect of collecting data for the empirical study.

2.1.6. Summary and Associated Problems

The use of integrated models is common practice in ADs. The analysis shows that integrated models play an important role in the three case studies. The ADs from our case studies contain a lot of repetition. A large part of this repetition is found in integrated models. Moreover, a large part of the elements needing updating for change scenarios are present in integrated models, and thus are repeated updates. Finally, we observed that integrated models are a source of inconsistencies.

In summary, we identified the following problems:

P1 Too much effort is spent in specifying repeated elements.

P2 For each change, too much effort is spent in repeated updates to elements.

P3 Manual updates with repetition in multiple models without explicit information on how models are related is error-prone and a breeding ground for inconsistencies.

2.2. Goal

In our work, we aim at automated composition of integrated architectural models. When using composition, an architect no longer specifies the integrated models directly, but instead specifies a composition that leads to the same integrated model. The composition, and the integrated model it defines, become an integral part of the AD.

The high-level goals of our approach for supporting composition of architectural models are:

Language support for compositions and relations
Support in ADLs is needed for rigorously specifying the *relations* between the models and what models must be integrated (a *composition specification*). Embedding both concepts in ADLs makes them integrals part of any AD. This ensures unambiguous specifications and facilitates consistent updates.

Guarantees about semantic preservation
Guarantees are needed about a composition function preserving the semantics of the models and relations being composed. This ensures that the meaning of the AD does not change because of composition.

Automated composition
Tool support is indispensable for making composition useful in practice. Automated composition allows an architect to easily obtain a composed model, to get an overview of several models, to see how models are integrated or to reveal possible conflicts between models.

3. Background

In this section we discuss the TMS case study in more depth and present the basic formal ADL. Both the case study and basic ADL are used in section 4 to explain our composition approach.

3.1. Case study: A Traffic Management System

The TMS is part of an industrial project involving eight companies, two universities (including the KULeuven) and the Flemish government. The system aims at flexible TMSs to cope with changing road circumstances, such as road works, traffic diversions and emergency clearances.

Table 2: Overview of views and models in the TMS

View	Purpose	Models	Fig
Top-level	End-to-end architecture	Overview (C&C)	4,6
Monitoring	Detectors and data fusion	Detectors (C&C)	
Data	Validation and storage	Data (C&C)	6
Traffic	Interpretation	Traffic (C&C)	
Actuator	Signalization	Actuator (C&C)	6
Deployment	Deployment on infrastructure	MultiAS (Deploy) SingleAS (Deploy)	5

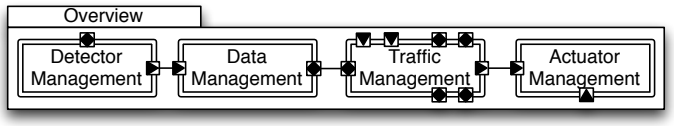


Figure 4: The Overview architectural model, key as in figure 3.

Table 2 briefly explains the purpose of the views and models of the TMS. The Overview model (figure 4) contains an end-to-end view on the system. The system monitors traffic using devices such as cameras and inductive loops to count the number of cars with the DetectorManagement component. Monitoring information is verified and collected in a repository in the DataManagement component. The monitoring information is used to control traffic in the TrafficManagement component through actuators such as traffic lights and signalization boards in the ActuatorManagement component.

The Detectors, Data, Traffic and Actuator models describe a substructure of the DetectorManagement, DataManagement, TrafficManagement and ActuatorManagement components respectively. The two deployment models in the Deployment view describe two possible deployment alternatives.

3.2. A Basic ADL

In this section we introduce a basic ADL that formalizes common architectural concepts. This includes C&C models and two types of models to support allocating the components to an infrastructure: infrastructure models and deployment models. The basic ADL serves as a basis to formalize composition in an ADL neutral way in the next section.

Set theory is used for specifying the ADL. Set theory is well known, easy to read and understand, and provides enough rigor for unambiguous definitions. In this section we only define the sets. Well-formedness rules are used to exclude invalid and redundant tuples from set-based specifications and are included in Appendix A.3. To prove properties of compositions, we also made an extended specification in Haskell (Boucké (2008)).

3.2.1. Views and Models

The basic ADL defines views and models as proposed by ISO 42010. The AD has a name and a set of views. Each of the views has a name and a set of models. In this basic ADL, the set of models consists of C&C models, infrastructure models and deployment models. Formally:

$$\begin{aligned} AD &= ID \times \mathbb{P}\, VIEW && \text{AD} \\ VIEW &= ID \times \mathbb{P}\, MODEL && \text{Architectural views} \\ MODEL &= CCMODEL \cup INFRA \cup DEPLOY && \text{Architectural models} \\ ID & && \text{Set of names} \end{aligned}$$

The next sections define the types of models in more details.

3.2.2. C&C models

An example of a component and connector (C&C) model is shown in figure 4. C&C models describe runtime software elements such as components, connectors and their interfaces, and links that describe the pathways of interactions. Components can have substructures, in which case *interface mappings* are used to map an outer interface (on the enclosing component or connector) onto an inner interface (on the internal element). Interfaces have a direction to indicate the information flow: in, out or inout. The sets defining a C&C model are[1]:

Example well-formedness rule for a C&C model are: "all elements have a unique name", "links in a C&C model must be between interfaces of components or connectors in the same model" and "connectors must be in between components, i.e. a link to an interface of a connector always comes from an

[1]The following abbreviated notations apply. All elements are referred to by their type with the instance name as subscript, e.g. a component C is referred to comp_C. Since the name is already shown in the subscript, the name is not shown in the tuple. We also use an abbreviated notation to refer to the internals of an element. For example, to refer to the interfaces of a component comp_C we use the notation comp_C.ints. All abbreviated notations are included in Appendix A.1.

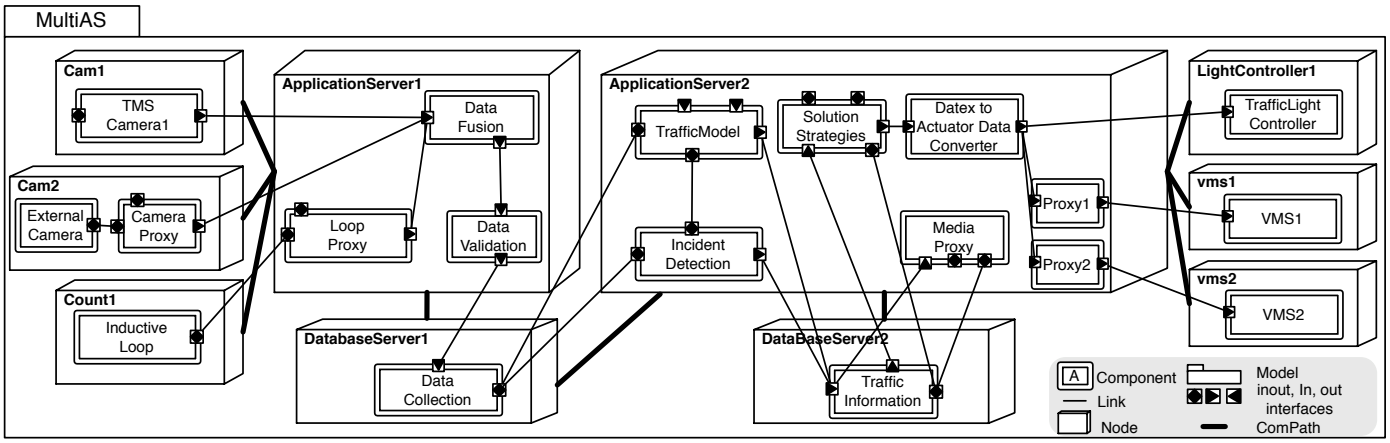


Figure 5: The MultiAS deployment model. MultiAS stands for multiple application servers.

$DIR = \{in, out, inout\}$	Directions
$INT \subset ID \times DIR$	Interfaces
$CC \subset ID \times \mathbb{P}\, INT \times SUB$	C&Cs
$COMP \subset CC$	Components
$CON \subset CC$	Connectors
$LINK \subset INT \times INT$	Links
$IM \subset INT \times INT$	Interfacemappings
$SUB \subset \mathbb{P}\, COMP \times \mathbb{P}\, CON \times \mathbb{P}\, LINK \times \mathbb{P}\, IM$	Substructures
$none = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle; \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \in SUB$	No substructure
$CCMODEL \subset ID \times \mathbb{P}\, COMP \times \mathbb{P}\, CON \times \mathbb{P}\, LINK$	C&C models

interface of a component". Appendix A.3.1 contains the complete set of well-formedness rules.

Using this definition of C&C models, we can now formally specify the Overview model of our case study, shown in figure 4:

$ccmodel_{Overview} = \langle comps, \emptyset, links \rangle$
$comps = \{comp_{DetectorManagement}, comp_{DataManagement}, \dots\}$
$comp_{DetectorManagement} = \langle \{i_{request}, i_{rawdata}\}, none \rangle$
$i_{request} = \langle inout \rangle$
$i_{DetectorManagement.rawdata} = \langle out \rangle$
$comp_{DataManagement} = \langle \{i_{rawdata}, i_{query}\}, none \rangle$
$i_{DataManagement.rawdata} = \langle in \rangle$
$i_{query} = \langle inout \rangle$
...
$links = \{l_1, ...\}$
$l_1 = \langle i_{DetectorManagement.rawdata}, i_{DataManagement.rawdata} \rangle$
...

The extract focusses on two components, DetectorManagement and DataManagement, with their interfaces and one link that connects the two components. We assume that all elements have a fully qualified name. For example, the two 'rawdata' interfaces can be distinguished by using the name of the component to which they belong as preposition.

3.2.3. Infrastructure and Deployment Models

To allocate components on a run-time infrastructure we introduce two additional models: an infrastructure model and a deployment model. Infrastructure models are used to document the infrastructure available to the software system before deciding on deployment. The deployment model extends the infrastructure model and contains the deployment configuration of components on the infrastructure. The concepts used in both models are based on UML deployment models.

An *infrastructure model* defines nodes and their connections through communication channels. A *node* represents a computer system, and can be annotated with several properties to describe its characteristics. Example nodes are embedded devices, servers and personal computers. A *communication channel* represent a connection between the nodes, and also has properties. Example communication channels are local area networks (LAN), wireless local area networks (WLAN) and GPRS connections.

A *deployment model* combines an infrastructure model with a C&C model to allow the allocation of components and connectors to nodes and channels. A deployment model can be considered as an integrated model[2], as it combines both elements of the infrastructure and C&C models. Figure 5 shows an

[2]This roughly corresponds to the combined view of Clements et al. (2003), which is a view that is a combination of styles. In the terminology of this paper this comes down to combining the types of several models.

example deployment model for the TMS called MultiAS.

The formal definition of infrastructure and deployment models is given below. We left out properties in this formal definition because they are not important for the remainder of this paper.

$NODE$	Set of Nodes
$COMPATH \subset ID \times \mathbb{P}\ NODE$	Com. channels
$INFRA \subset ID \times \mathbb{P}\ NODE \times \mathbb{P}\ COMPATH$	Infrastructure models
$DNODE \subset NODE \times \mathbb{P}\ COMP \times \mathbb{P}\ CON \times \mathbb{P}\ LINK$	Deployment nodes
$DCOMPATH \subset ID \times \mathbb{P}\ NODE \times \mathbb{P}\ LINK$	Com. channels
$DEPLOY \subset ID \times \mathbb{P}\ DNODE \times \mathbb{P}\ DCOMPATH$	Deployment models

An example of a well-formedness rule is " a communication path in an infrastructure model or deployment model should be between nodes in the same model". The well-formedness rules that apply to infrastructure and deployment models are described in Appendix A.3.2.

3.3. Relations

Relations specify how models are related before they are composed (Boucké et al. (2008); Sabetzadeh et al. (2006)). Boucké and Holvoet (2008) already present several types of relations, of which we use the following:

- **Unification**: expresses that two elements (either components or connectors) that appear in two different C&C models are the same element. Interface unifications express which interfaces are the same.
- **Submodel**: expresses that a C&C model describes the internal structure of a component or connector of another C&C model. Interface mappings are used to map internal elements on the enclosing element.
- **Allocation**: expresses that a set of components and connectors is allocated onto a specific node in an infrastructure model.

Relations can be specified either by a software architect or automatically deduced by tools using heuristics.

The following sets define the relation types:

Examples of well-formedness rules are: "a component can only be unified with a component", "one can only unify components or connectors from different models". The complete set of well-formedness rules can be found in Appendix A.3.3.

$IU \subset ID \times INT \times INT$	Interface unifications
$UNIF \subset ID \times CC \times CC \times ID \times \mathbb{P}\ IU$	Unification relations
$SUBMODEL \subset ID \times CC \times CCMODEL \times \mathbb{P}\ IM$	Submodel relations
$ALLOC \subset ID \times NODE \times \mathbb{P}\ CC$	Allocation relations
$REL = UNIF \cup SUBMODEL \cup ALLOC$	The set of all relations

The following definition extends our basic ADL with support for relations between models:

$$AD' = ID \times \mathbb{P}\ VIEW \times \mathbb{P}\ REL$$

In this paper we do not distinguish relations between models in a single view and relations between models in different views.

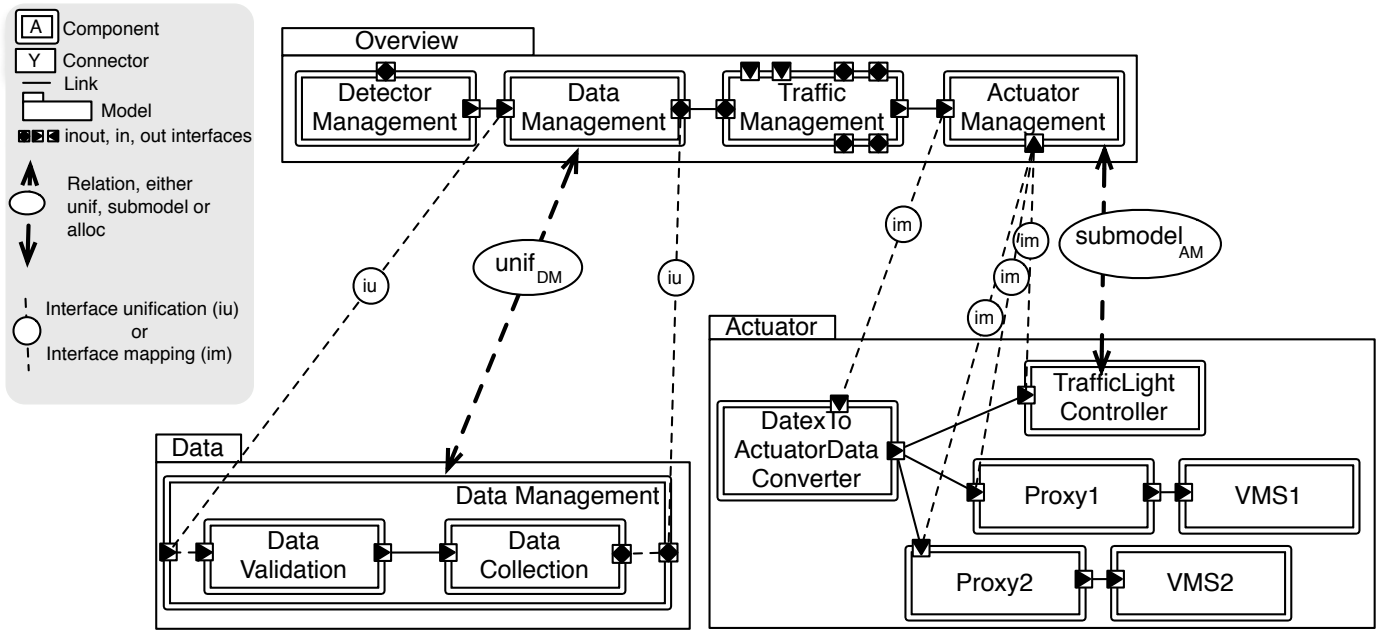
3.3.1. Example of Unification and Submodel

Using the above definitions, we can explicitly specify the relations between the models needed for composition. As an example, figure 6 shows a unification relation between the Overview model and the Data model and a submodel relation between the Overview model and the Actuator model.

We briefly explain the new models. The Data model contains the DataManagement component which is decomposed in: DataValidation to validate monitoring data and DataCollection to store data and allow queries. The Actuator model contains the DatexToActuatorDataConverter component to translate commands to low-level control signals for actuators. The other components represent actuators, namely the TrafficLightController and two variable message signs (VMS) with a proxy to translate the commands to the appropriate format understandable for a VMS.

The unification relation expresses that the DataManagement component of the Overview model is the same as the DataManagement component of the Data model. Interface unification specifies which interfaces are the same.

The submodel relation expresses that the Actuator model describes the substructure of the ActuatorManagement component. Interface mappings specify which interfaces of ActuatorManagement map on interfaces in the Actuator model.



$unif_{Data} = \langle comp_{Overview.DM}, comp_{Data.DM}, DM, ius \rangle$
$ius = \{iu_1, iu_2\}$
$iu_1 = \langle i_{Overview.DM.Provider}, i_{Data.DM.Provider} \rangle$
$iu_2 = \langle i_{Overview.DM.Query}, i_{Data.DM.Query} \rangle$

$submodel_{AM} = \langle comp_{ActuatorManagement}, ccmodel_{Actuator}, ims \rangle$
$ims = \{im_1, im_2, im_3, im_4\}$
$im_1 = \langle i_{ActuatorManagement.DatexActuator}, i_{DatexToActuatorConverter.DatexActuator} \rangle$
$im_2 = \langle i_{ActuatorManagement.ActuatorOperator}, i_{TrafficLightController.Operator} \rangle$
$im_3 = \langle i_{ActuatorManagement.ActuatorOperator}, i_{Proxy1.Operator} \rangle$
$im_4 = \langle i_{ActuatorManagement.ActuatorOperator}, i_{Proxy2.Operator} \rangle$

Figure 6: Example of unification and submodel relations. Top: visual notation. Bottom: formal notation.

4. Composition of Architectural Models

The basic ADL provides the necessary building blocks for specifying composition. Composition is introduced in four steps. We start by discussing how composition is specified and embedded in the ADL. Then, we provide two examples of how composition is used in the TMS. Next, we discuss the formal specification of the composition function and the semantics of composition. Finally, we instantiate composition in xADL and its tool to demonstrate the feasibility of the approach.

4.1. Language Support for Composition

A composition is the integration of several architectural models based on the relations defined between these models. We distinguish between a composition specification and a composition function.

A *composition specification* has a name and contains the models and relations to be composed. This is captured in the following formal definition:

$$COMPSPEC \subset ID \times \mathbb{P}\, MODEL \times \mathbb{P}\, REL \quad \text{Set of composition specs}$$
$$spec_{name} = \langle inModel, inRels \rangle \quad \text{Naming conventions}$$

We use the terms 'input models' (inModel) and 'input relations' (inRels) for models and relations in the composition specification. Well-formedness rules ensure that the input relations are defined on the input models, and that there is no interference between the relations. The complete set of well-formedness rules is given in Appendix A.3.4.

The *composition function* takes a composition specification as input and defines an integrated model together with a set of traces. This leads to a function with the following signature:

$$composition : COMPSPEC \rightarrow MODEL \times \mathbb{P}\, TRACE$$

In section 4.3, we revisit the formal specification of the composition function. The term 'integrated model' is used to refer to the model defined by composition. The term 'traces' denote the resulting set of

traces.

Traces describe the relations between input and integrated models. Traces are also a kind of relation between models, but they are always the result of applying a composition and that is why they are treated separately. Formally, traces are defined as:

$$TRACES = (MODEL \times MODEL) \cup (COMP \times COMP) \cup (CON \times CON) \cup (INT \times INT) \cup (NODE \times DNODE) \cup (COMPATH \times DCOMPATH)$$
$$trace = \langle out, in \rangle$$

With this definition of composition, we can extend the basic ADL with support for compositions.

$$AD'' = ID \times \mathbb{P}\, VIEW' \times \mathbb{P}\, REL$$
$$VIEW' = ID \times \mathbb{P}\, MODEL'$$
$$MODEL' = CCMODEL \cup INFRA \cup DEPLOY \cup COMPSPEC$$

A composition specification is added to the set of models. An architect obtains the integrated model by applying the 'composition' function on the specification. The resulting integrated model is a regular model that can be used for visualization, analysis or in another composition.

4.2. Using Compositions

First, we illustrate how an architect can use composition for integrated models before going into the formal details of the composition function in 4.3. Composition is illustrated using two examples: an example of composing C&C models and a deployment model.

4.2.1. C&C Model Composition

Figure 7 shows our running example. Remember that an architect wants to compose the models above the arrow with each other, resulting in the integrated model below the arrow.

Specifying a composition involves two steps. Firstly, the architect specifies the relations between the models that need to be composed. Potentially some or all of the relations are already present in the AD for other compositions or could be specified for other reasons such as consistency analysis. In that case, an architect can reuse the existing relations. We already specified the unif_{Data} and submodel_{AM} relations between these models in figure 6.

Secondly, the architect specifies the composition. This is shown below:

$$spec_{Integrated} = \langle \{ccmodel_{Overview}, ccmodel_{Data}, ccmodel_{Actuators}\}, \{unif_{Data}, submodel_{AM}\} \rangle$$

The specification states that the Overview, Data and Actuator model are composed using the unif_{Data} and submodel_{AM} relations.

The integrated model itself is the result of applying the composition function. This could be the responsibility of an architectural design tool that applies this function on the specification as shown below:

$$(ccmodel_{Integrated}, traces) = composition(spec_{Integrated})$$

The result is the Integrated model below the arrow in figure 7. The unif_{DM} relation leads to an unified component DataManagement with DataValidation and DataCollection as elements of its substructure. The submodel_{AM} relation defines that the components of the actuator model are in the substructure of the ActuatorManagement component. The remainder of the elements are retained unchanged from the input models. The traces between the input models and the integrated model called traces are shown as light-gray lines. The traces between the interfaces are left out to keep the figure understandable.

4.2.2. Deployment Model

An alternative to describing a deployment model directly is to define it in terms of a composition. The use of composition for deployment is similar to C&C composition. Figure 8 contains an example of allocating the components of the Actuator model to the InfraSingle infrastructure model. The latter model contains an application server and several nodes to allocate the actuator devices (LightController1, vms1 and vms2). This is a simplified example of an infrastructure model for the TMS.

We briefly discuss the two steps required for composotion. Firstly, the architect defines the relations between the models. The example contains four allocation relations, allocating six components in total. For example, alloc_{AS} allocates the $\text{comp}_{DatexToActuatorDataConverter}$ and two proxies to

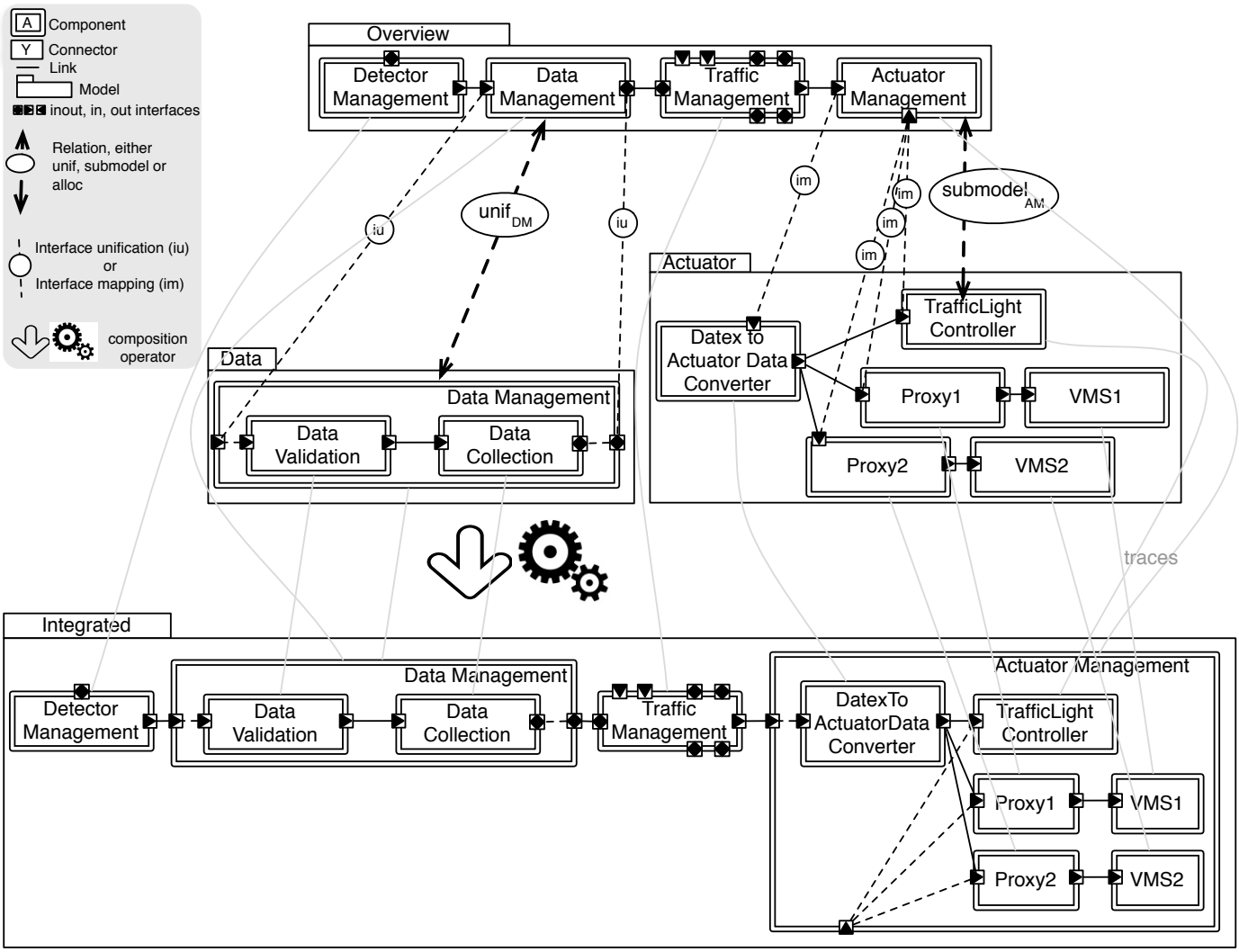


Figure 7: Example composition, defined by the $\text{spec}_{Integrated}$.

$\text{node}_{ApplicationServer}$.

$alloc_{AS} = \langle node_{ApplicationServer}, \{comp_{DatexToActuatorDataConversion}, comp_{Proxy1}, comp_{Proxy2}\}\rangle$
$alloc_{LC1} = \langle node_{LightController}, \{comp_{TrafficLightController}\}\rangle$
$alloc_{vms1} = \langle node_{vms1}, \{comp_{VMS1}\}\rangle$
$alloc_{vms2} = \langle node_{vms2}, \{comp_{VMS2}\}\rangle$

Secondly, the architect defines the composition specification as shown below. The composition specification comprises two models and four relations.

$inModel = \{ccmodel_{Actuator}, infra_{InfraSingle}\}$
$inRels = \{alloc_{AS}, alloc_{LC1}, alloc_{vms1}, alloc_{vms2}\}$

$spec_{deploy1} = \langle inModel, inRels, SimpleSingleAS\rangle$

The integrated model called SimpleSingleAS is obtained by applying the composition function on this specification. The result is shown below the arrow in figure 8. It is a simplified version of the SingleAS model of the TMS.

4.3. Specification of the Composition Function

In this section we show an extract of the definition of the composition function in figure 9. We only show the details on how C&C models with a unification relation are composed, the other parts of the composition function are similar. The full definition of the composition function is given in Boucké (2008).

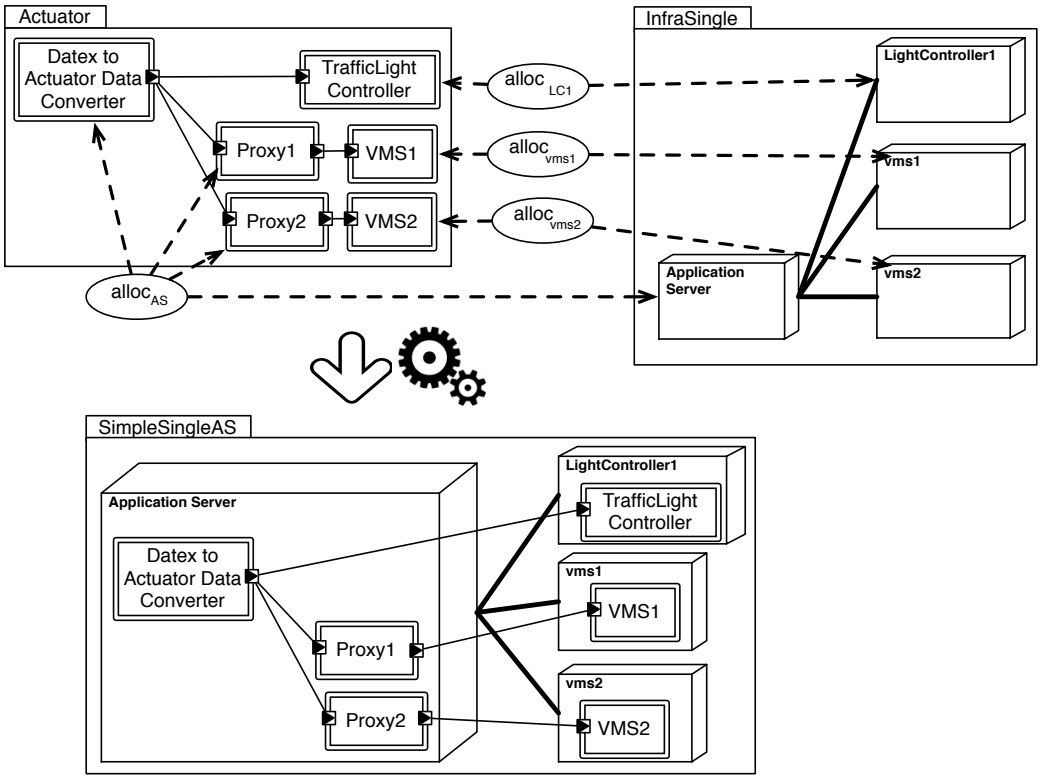


Figure 8: Example composition, defined by $\text{spec}_{deploy1}$. SimpleSingleAS stands for simplified example of the single application server model (SingleAS model).

We describe the most important functions one by one, progressively focussing on unification of C&C models.

We begin with the description of the composition function (figure 9 a). There are three possible cases. Firstly, the composition function is used to compose several C&C models. This is handled in the cccompose function and results in a C&C model. Secondly, the composition function is used to allocate C&Cs to an infrastructure. This is handled in the deployment function, resulting in a deployment model. Thirdly, a composition can be a combination of both previous cases. This is handled by the mixedCompose function using both other functions.

Next, we focus on the cccompose function to compose C&C models (figure 9 b). This function is build up around three sets related, nonRelated and links. The set related represents the components implied by relations with the relatedElements function. The set nonRelated defines the C&C elements for which there is no relation in the composition specification. Finally, the set links represents the links in the integrated model defined by the linker function.

Two small help functions are the allCC function and tt function. The allCC function maps an element or a set to all C&C that are recursively contained in the element or set. The tt function maps an element to an element with an updated qualified name (because the elements will be part of a different model) and traces between this element and the input models. The tt function is used when elements in the input and output model only differ in their name. This is the case for the nonRelated set.

We focus on the elements defined by relations. The relatedElements function (figure 9 c) is specified recursively in terms of the relatedElement (figure 9 d), which in turn is specified in terms of a help function for each type of relation, i.e. the submodel and unifiedElement functions. The ++ operator specifies a new tuple that is the pairwise union, i.e. the first element in the output tuple is the union of the first elements in the input tuples, etc.

We describe the unifiedElement function (figure 9 f) in more details. This function consists of four main parts. The first part defines the unified interfaces and their traces (the unifiedInterfaces function).

Figure 9: Formal definition of the composition function.

The second part defines the interfaces that are not unified (nonUnified). The third part defines the substructure of the unified element (sub). This substructure is taken over from one of the elements. Well-formedness rules on the composition specification make sure that these substructures are compatible with each other (rule A.11 in Appendix A.3.3). The final part defines the traces.

The unifiedInterfaces function (figure 9 g) is specified recursively in terms of the unifiedInterface function (figure 9 h) which defines the result of a single interface unification. The unifiedInterface function defines a unified interface and two traces to the interfaces in the input models.

4.4. Semantics of Composition

The formal specification of model composition allows us to specify the semantics of the composition function. The semantics are specified as rules on the input and output of a composition. Together these rules determine the outcome of a composition. In this section we discuss the most important rules that define the semantics of composition. The full set of rules can be found in Appendix A.

4.4.1. Composition Results are Traceable

The results of a composition are traceable to the input models. An example rule states that, for each component and connector in the integrated model, there exists a corresponding component or connector in the input models, identifiable through traces. This rule is shown below. Similar rules are defined for interfaces, nodes and communication paths.

$$\forall c_{out} \in allCC\ outModel, \exists model \in inModels, \exists c_{in} \in allCC\ model : \langle c_{out}, c_{in} \rangle \in traces$$

4.4.2. Composition Respects Relations

A composition should correctly reflect the relations selected in the composition specification. For example, when there is a unification relation between two components, we expect the integrated model to have a component that corresponds to these two input components. The following rule formally defines the semantics of a unification:

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

The 'allInts' function recursively returns all interfaces. The 'int' function returns all interfaces, non-recursively. The first part of the rule (lines 2-3) captures the semantics of a unified component or connector. More specifically, line 2 expresses what is present (positive) and line 3 expresses what cannot be present (negative). A similar pattern is followed in the lines that follow. The second part (lines 4-6) captures the semantics of unified interfaces. The last part (lines 7-8) captures the semantics for non-unified interfaces. Similar rules are defined for the submodel and allocation relations.

4.4.3. Composition Preserves the Model Semantics

A composition should not affect or change the meaning of the input models. The definition of the composition function implies that the integrated model is well-formed, because it is an element of the MODEL set.

Element distinction. Model composition preserves the distinction between elements defined in the input models. For example, if an input model contains components A and B, the integrated model cannot contain a single component that represents both A and B. This would imply that the composition does not correctly reflect the distinction between components A and B in the input model. The same holds for interfaces, nodes and communication paths. This is captured in the following formal statements:

$$\forall c_{out} \in allCC\ outModel, \forall model_1, model_2 \in inModels,$$
$$\exists c_1 \in allCC\ model_1, \exists c_2 \in allCC\ model_2 :$$
$$(\langle c_{out}, c_1 \rangle \in traces \wedge \langle c_{out}, c_2 \rangle \in traces \wedge c_1 \neq c_2) \Rightarrow model_1 \neq model_2$$

Substructures. A composition preserves substructures. If an element is part of a substructure in the input models, it should still be part of the substructure in the integrated model. This is captured in the following rule:

$$\forall c_{in} \in inModels.ccs, \forall c_{insub} \in c_{in}.sub.css, \exists c_{out} \in outModel.ccs,$$
$$\exists c_{outsub} \in c_{out}.sub.ccs : \langle c_{outsub}, c_{insub} \rangle \in traces) \Rightarrow \langle c_{out}, c_{in} \rangle \in traces \wedge$$
$$(\nexists c_{o2} \in outModel.ccs : \langle c_{o2}, c_{insub} \rangle \in traces)$$

The 'ccs' function returns components and connectors, non-recursively. The rule 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 is a trace between c_{in} and c_{out}. This rule ensures that the element is part of the same substructure in the integrated model. The last line 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 integrated model.

Link preservation. Composition preserves links between the elements. If elements are linked in the input models, they are linked in the integrated model.

$$\forall model \in inModels, \forall i_{in1}, i_{in2} \in (allInts\ model),$$
$$\exists! i_{out1}, i_{out2} \in (allInts\ outModel) :$$
$$\langle i_{out1}, i_{in1} \rangle \in traces \wedge \langle i_{out2}, i_{in2} \rangle \in traces \wedge connected\ i_{in1}\ i_{in2}$$
$$\Downarrow$$
$$connected\ i_{out1}\ i_{out2}$$

The rule states that for each connection between interfaces in the input models, there exists a connection between the corresponding interfaces in the integrated model.

4.4.4. Composition is Complete

The result of a composition is complete, i.e. all elements from the input models can be traced to a unique element in the integrated model. An example rule for components and connectors is shown below.

$$\forall model_{in} \in inModels, \forall c_{in} \in allCC\ model_{in},$$
$$\exists! c_{out} \in allCC\ outModel : \langle c_{out}, c_{in} \rangle \in traces$$

5. xADL&Co: Instantiating Composition in xADL

As an illustration of the feasibility of composition, we extended the xADL language defined by Dashofy et al. (2005) and the associated development environment ArchStudio to support compositions.

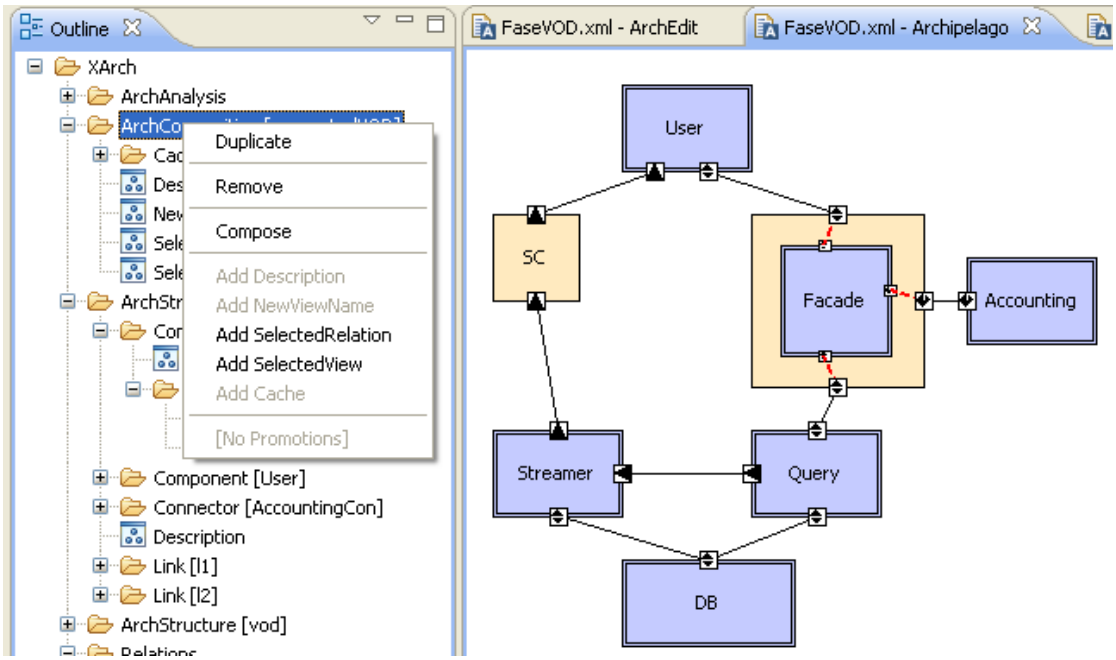


Figure 10: Snapshot of xADL&Co, the ArchStudio extension for composition.

5.1. *Extending xADL for Composition*

The extension of xADL for supporting composition manifests itself in several aspects.

Firstly, a *language extension* is needed that introduces additional models (infrastructure, deployment), relations (unification, submodel, subelements) and composition specifications. Under the hood this requires several additionl XML Schemas that extend the xADL language definition.

Secondly, a tool is needed to support *automated composition*. Such tool support is indispensable to make model composition practical. Tool support allows architects to quickly obtain a unified perspective on the architecture and reveal conflicts between models. Under the hood, the tool is an Eclipse plugin for ArchStudio.

Finally, we extended the *visual tool* to show substructures, infrastructure and deployment models. Figure 10 shows a snapshot of the tool.

5.2. *The Role of Formalization*

xADL itself is not supported by a formal language. Language concepts are defined in XML schemas to allow flexible and fast prototyping of language definitions.

The main role of our formalization is providing an unambiguous description of relations and compositions. The formalization helps to guide the designers and developers in building a consistent and conceptually sound tool. We also formalized the basic elements of an ADL, thus providing a formal basis for the part of the xADL language used by our tool.

Especially the introduction of traces during formalization proved to be a very useful for building the tool. During composition, the set of traces contains an up-to-date mapping between the integrated model and input models of all elements that are already processed by the composition function. This allows to navigate back and forth between input and output to check whether elements are already processed during composition.

5.3. *Limitations of the Tool*

The current tool illustrates the feasibility, but several key challenges remain for making it usable in an industrial setting. The main challenges are: (1) a visual and intuitive interface for specifying relations and composition; (2) automatic triggering of compositions when an input model changes; (3) automatic deduction of trivial relations using simple heuristics such as names or structures as done by Abi-Antoun et al. (2008); and (4) automatic compatibility checks of relations between models.

6. Empirical Evaluation

The goal of the empirical study is to evaluate consequences and tradeoffs of using composition. The evaluation builds on the study in section 2.1. It compares the original AD for the three case studies (AGVTS, TMS, DPS) with using an alternative AD including relations and compositions.

6.1. Empirical Procedure

6.1.1. Hypotheses

The focus of the evaluation is on differences in the ADs. The evaluation does not include usability or an evaluation of the architectural tools itself (further discussed in section 6.4). In addition to the three hypotheses mentioned in section 2.1.1, we add two hypotheses on the use of relations and compositions. The fourth hypothesis is that the number of elements that must be specified by the software architect (manual specification) is smaller for the AD with relations and composition than the original AD (H4). The fifth hypothesis is that in the event of changes, a software architect has to change less elements in the architectural description (H5).

6.1.2. Constructing an Alternative AD

As part of this empirical study, we constructed an alternative AD using relations and compositions. We made sure that the *architecture* of the original and alternative AD were the same, only the *description* differs. The alternative AD contains at least the models present in the original AD. Only information is added, such as additional models, relations and compositions specifications. As a final check, we involved the original architects to confirm that the AD still described the same architecture. The following steps were followed during construction of the alternative AD.

1. Two of the three ADs had no infrastructure models. We extracted the infrastructure model from the existing deployment models and added this to the AD.
2. The AGVTS contained several models that are nearly an integrated model, but also contained some new information. In these cases, we extracted the new information in a separate model to allow the use of compositions.
3. None of the original ADs had relations that were precise enough for automated composition. We added unification, submodel and allocation relations between the existing models without altering them.
4. In the last step, we replaced the combined C&C models and the deployment models with composition specifications.

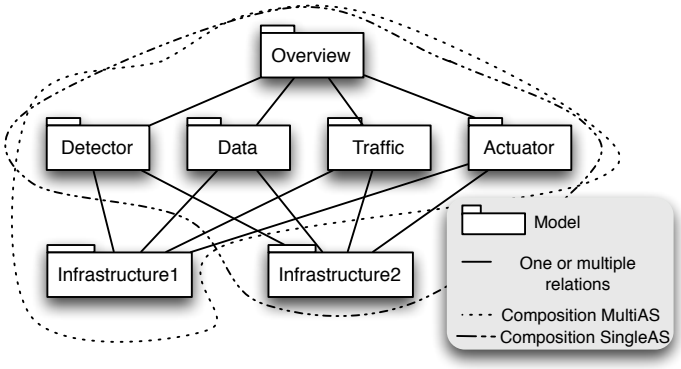


Figure 11: Overview of relations and compositions in the TMS.

As an example, we summarize the results for the TMS case study in figure 11. The first step introduces two infrastructure models, extracted from the two existing deployment models MultiAS and SingleAS. The second step adds four unifications and eighteen allocations. There is one unification relation for each component in the Overview model, related to the respective components in the Detector, Data, Traffic and Actuator model. There is an allocation relation for each node in the infrastructure models, allocating in total thirty-eight components. The last step replaces the two deployment models with two composition specifications. Since the relations are already specified, the composition specifications are relatively simple. Both composition specifications are given below. They differ only in the selection of the infrastructure model.

$$spec_{multi} = \langle \{ccmodel_{Overview}, ccmodel_{Detectors}, ccmodel_{Data}, ccmodel_{Traffic}, ccmodel_{Actuators}, infra_{Infrastructure1}\}, \emptyset, MultiAS \rangle$$

$$spec_{single} = \langle \{ccmodel_{Overview}, ccmodel_{Detectors}, ccmodel_{Data}, ccmodel_{Traffic}, ccmodel_{Actuators}, infra_{Infrastructure2}\}, \emptyset, SingleAS \rangle$$

6.2. Results

This section discusses the results of investigating the hypotheses. At the end of this section we make a remark on inconsistencies and size.

6.2.1. Hypothesis 4: Manual Specification

During problem analysis we identified the potential for composition (section 2.1.3). Here, we investigate to what extent automatic composition can counterbalance the cost of specifying relations and compositions. As we only have the AD available for inspection, we used the number of elements that must be specified by the architect as an indicator of this cost. This includes all architectural elements, relations and composition specifications, but without the elements in the integrated models as these elements are automatically derived by a tool.

Result. Figure 13 summarizes the results. It shows that in three cases there is a significant difference in the number of elements that require manual specification. For the TMS and AGVTS case studies, this difference is the largest (29%). For the DPS case study the difference is smaller (11%), but still significant. This confirms hypothesis H4.

The difference is smallest for the DPS case study because about half of the repetition is not caused by integrated models, but by separate features (the last cause in section 2.1.3).

6.2.2. Hypothesis 5: Change Scenarios

Next, we analyze to what extent relations and compositions help in handling changes. We focus on the same change scenarios as in the problem analysis in section 2.1.4, and measure the effort of changing the description in the same way.

Each of the scenarios is applied to both versions of the AD. For each of the scenarios, we counted the number of elements that an architect manually needs to add, to remove or to change the AD. For the AD with relations and compositions we also counted the total number of changes to relations and compositions.

Result. The results are shown in figure 14. The abbreviation RTMS refers to the version of the AD with relations and composition. The numbers with a plus represent the number of changes needed for a particular scenario. The results show that the number of changes is between 42% and 52% lower for the AD with relations and compositions. This confirms hypothesis H5.

	S1	S2	S3	S4
TMS	+21	+26	+33	+103
RTMS	+11	+13	+14	+54
%difference manual	52%	50%	42%	52%

Figure 14: Overview of changes required for the different scenarios.

Note that the scenarios not only triggered changes to the models, but also to the model elements. For example, in S1 and S2, allocation relations must be added. In S3 a submodel relation must be added for the Data model. In S4 we had to add an allocation relation for each node in the new deployment alternative.

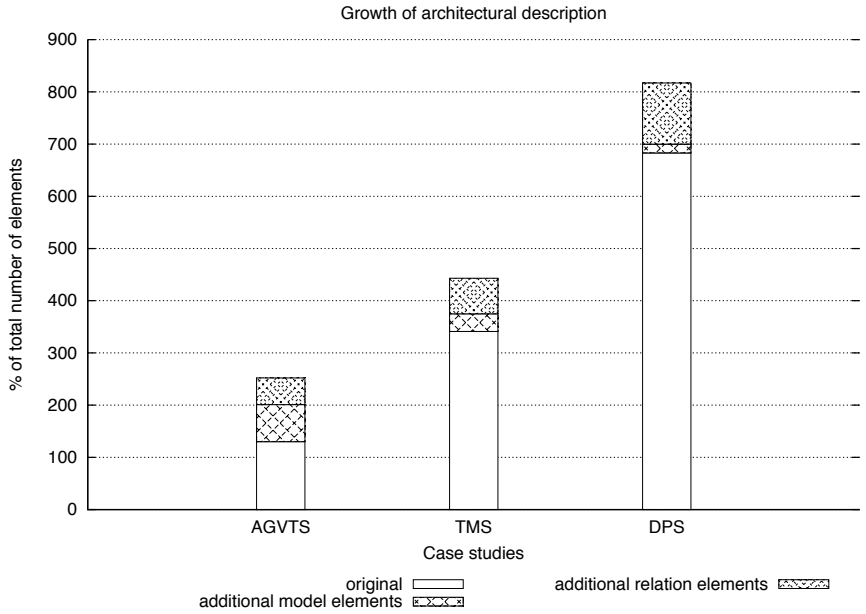
6.2.3. A Remark on Inconsistencies and Size

During the problem analysis we observed that most inconsistencies are in the integrated models. Replacing integrated models with automated composition prevents inconsistencies in integrated models. This corresponds with problem P3 (see section 2.1.6).

Another remark is that the total AD will be larger in size. The total size includes the elements in all models (this explicitly includes integrated models), the elements in all relations and compositions, and all elements in the integrated models of composition. Restructuring the AD for use with relations may add additional model elements, and describing relations and composition will also add to the size. The AD may be more complete and less ambiguous, but the complete document will be larger.

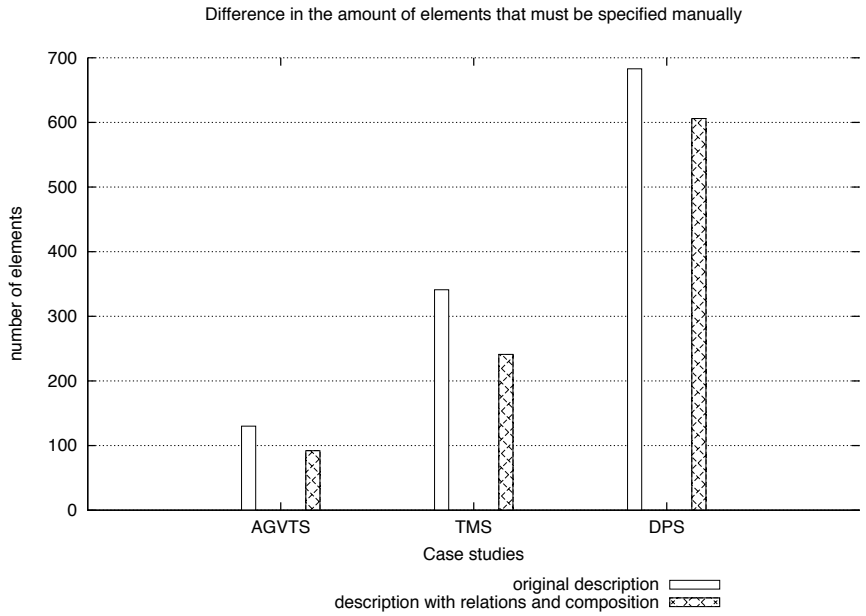
To quantify this, we compared the size of two versions of the AD using the total number of elements in the description as an indicator. Figure 12 summarizes the size of the ADs. The left hand side shows the size in a stacked histogram. The shaded region shows the information added. We also studies which elements can be attributed to models or to relations/compositions. The right hand side of figure 12 summarizes the absolute values and percentages. This reveals a significant increase in size of the alternative AD between 20% and 94%.

The significantly higher size of the AGVTS AD



	#models	#model elements	#relation elements	#total	%increase in size
AGVTS	+5	+71	+52	+123	94%
TMS	+2	+34	+68	+102	30%
DPS	+1	+17	+117	+134	20%

Figure 12: Illustration of the increase in size of the alternative AD. Left: Stacked histogram. Right: Absolute values and percentages.



	#manual for original AD	#manual for rel&comp	%less compared with original
AGVTS	130	92	29%
TMS	341	241	29%
DPS	663	606	11%

Figure 13: A comparison between the amount of elements that must be specified manually for the original AD and the AD including relations and composition. Left: Histogram. Right: Absolute values and percentages.

has several causes. Firstly, the description contains only a few strongly related models. Secondly, there is more restructuring needed to allow the use of relations and compositions. For example, there are several models that are nearly integrated models but contain one or more new elements. These models are split into an integrated model and a model with new information by the second step of constructing the alternative AD. The other two case studies required less restructuring and both have an increase of approximately 20%.

6.3. Summary of Analysis

The main results of this analysis, and their connections with the problems outlined in section 2.1.6, are:

- The architect must specify less elements for the alternative AD and for the change scenarios (corresponds to P1 and P2) because integrated models are automatically updated.
- The inconsistencies examined in the problem analysis are prevented in the alternative AD (corresponds to P3) because integrated models are automatically kept consistent with the models being integrated.
- The alternative AD is larger in size

The fact that both relations and compositions are rigorously captured in the AD made it easier to keep the AD consistent. This corresponds with P3.

Implications. An implication is that the architect must specify significantly less elements, and ends up with an AD that is more complete as it includes relations and traces to integrated models. The results also imply that an architect must change significantly less elements for change scenarios. This indicates that the cost of specifying relations and compositions is compensated by the benefits of automated composition.

One remark is that a potential reader who wants to study the complete AD must study significantly more elements because the alternative AD is larger. However, not all stakeholders will be interested in this, so the impact of the growth in size is probably limited.

6.4. Threats to Validity

In this section we briefly discuss a number of threats to the validity of the empirical study and discuss how we have tried to limit the impact of these threats.

6.4.1. Cases

A threat is the kind of ADs used in the study. Each of the ADs in the study is a cooperation between DistriNet and one or more industrial partners. They are representative for the ADs in DistriNet and for typical ADs build following a multi-model documentation approach by architects who are experienced in these kinds of approaches. A threat is that the results are generalized to other cases that are different in nature without consideration.

Another threat is that one of the authors of the study was involved in specifying the architecture for the AGV application. This is a threat as it could imply a certain bias. We believe that the impact of this is limited as the architectural description of this system was finished and available before the empirical study started, so the empirical study itself did not influence the architecture of the AGV system.

A next threat is that we had to prepare the original architectural description to allow a broader use of compositions as described in section 6.1.2. The preparation only involved extracting certain information in separate models, without altering the meaning of the description in any way. To confirm this we involved some of the original architects. This refactoring to improve the use of compositions could indicate a potential bottlenecks of the composition approach and needs further investigation.

Finally, only a limited number of cases is considered. Broader studies are required to confirm the findings.

6.4.2. Metrics

A threat is that the measurements are only based on the architectural description, not on the time or effort spend by software architects. The problem with measuring the effort of architects is that the quality of the tools to describe relations and compositions (needs a production ready tool!) and the knowledge of the software architects on relations and compositions will have a large influence on the results. As

a consequence we used measurements based on the ADs as an indicator of the possible effort spend by practicing architects. The threat to validity is that measurements based on the AD will probably not correspond one-to-one to the effort spend by a software architect, and thus the results must not be interpreted in an absolute manner but must be seen as an indicator.

Another threat is that we mostly had to defined our own metrics. The area on multi-view architectural descriptions is still relatively young and measurements on an architectural description are uncommon. Consequently, no previous studies or metrics existed that allowed to confirm or refute our hypotheses. E.g. we needed to quantify repetition, change and the number of elements specified in an architectural description. As a result, the metrics could be biased to our approach, or not provide practical information at all.

To provide some trust in the metrics, we preferred to keep them as simple as possible so that the results speak for themselves. Each of the metrics is based on the number of elements according to some property (repetition, changes, ...). For example, to quantify changes, we calculated the number of elements to be changed and the proportion of changes to be done in integrated models. The metrics to analyse the change scenarios is based on an existing metric proposed by Molesini et al. (2008).

Finally, the empirical study and the metrics used are narrowly scoped to investigate the use of relations and compositions for certain change scenarios. The broader influence of using relations and compositions needs further investigation.

6.4.3. Usability

The empirical evaluation does not include an evaluation of the usability of the xADL&Co tool. A threat to validity is that the usability of the extensions and tool will have a significant impact on the feasibility of using relations and composition in practice. As a consequence, some usability questions remain open. How easy can the extension be understood by architects? Do they over complicate the models by introducing additional information between models instead of in the models themselves? Do architects use the relational and composition techniques correctly?

Our experiences with applying these techniques revealed no real problems from this perspective. On the contrary, the proposed relations and compositions seem to align with the expectations of architects. But this needs deeper investigation once the research demonstration tool is further advanced to support a usability study involving a large group of architects.

7. Related Work

Our composition approach builds upon experience with model composition. It employs ideas from several other approaches in the domain of software architecture and ADLs. The distinguishing aspects of our work are: (1) the use of composition to manage repetition, changes and inconsistencies in integrated models on the architectural level; (2) the focus on architectural models such as C&C, infrastructure and deployment; and (3) the explicit embedding of composition in an ADL.

Architectural. To the best of our knowledge, there are no ADLs that support composition of architectural models. The tools associated with ADLs also offer no alternative solution to handle the problems associated with integrated models. There are, however, approaches using composition on the architectural level. We discuss several examples.

Moriconi and Qian (1994) did work on composition in the context of ADLs, in particular for correct composition of partial descriptions (abstract and concrete descriptions). That work focusses on refinement of architecture in general, not with the use of architectural models. The work is strongly related to our formal underpinning and our use of refinement.

Egyed et al. (Egyed (2000); Egyed and Medvidovic (2000); Medvidovic et al. (2003)) exploit redundancies between different types of UML diagrams to ensure consistency between the views. The underlying strategy is to transform and integrate several heterogenous models with each other to facilitate consistency checking. Examples include UML diagrams such as class, interaction and state diagrams. In contrast, our work is about composing several architectural models to support integrated models, and embedded this composition specification in an ADL.

Our work is not focussed on finding inconsistencies, but rather on composing architectural models to see how the elements are effectively integrated.

Grundy and Hosking (2003) propose SoftArch, an environment for flexible architecture modeling relying on successive refinement. Such refinement is essential to view the system on different levels of abstraction, with traceability support between the different levels. Refinement can be done in three ways: enclosing components in another component, adding a subview for an element and specifying explicit refinement links between elements. Furthermore, SoftArch also supports an extensible language, analysis tools and run-time visualization of systems. Our work differs by making relations first-class citizens of the ADL, and by using these relations for composition of architectural models.

Giese and Vilbig (2006) formally define composition of component behavior and properties of this composition. The focus on behavioral composition is complementary to our approach. Differences with are approach are that relations are not explicitly specified but matching is programmed in the composition algorithm. And more specifically for integrated models, it remains unclear if and how composition specifications are embedded in the AD.

Abi-Antoun et al. (2008) propose an approach for differencing and merging version of an AD in ACME (described by Garlan et al. (2000)). Important differences with our work are that the authors: (1) merge complete ADs, where we compose individual models; and (2) that we explicitly capture the relations and compositions *in the ADL* and support infrastructure and deployment models.

MDE. Although our work originates from an architectural background, it is closely related to research on Model Driven Engineering (MDE).

MDE focusses on the creation and use of models to create software and model transformation (transforming models to other models) to bridge levels of abstraction. The subclass of model transformations with which our composition approach is most closely related to is model merging as defined by Kolovos et al. (2006). Model merging encompasses comparison and merging of design models. There are existing approaches to merge class diagrams (Clarke and Baniassad (2005); Katara and Katz (2003); Rubin et al. (2008)), state diagrams (Nejati et al. (2007); Nejati (2008)) and interaction diagrams (Nejati (2008)), as well as combinations of the previous (Barais et al. (2008); Sabetzadeh et al. (2008)). Some approaches can be extended to any type of UML model, for example Fabro et al. (2006) and Kolovos et al. (2006), by defining relations and transformations on the UML meta-model level. Bendix and Emanuelsson (2009) and Mehra et al. (2005) provide support for differencing and merging diagrams for version control. Yet others define a generic approach for differencing and merging control (Mehra et al. (2005)), a reference process (Jeanneret et al. (2008)), a generic approach (Fleurey et al. (2008)), a canonical scheme (Bzivin et al. (2006)) and an algebraic view on the semantics (Herrmann et al. (2007)) of model merging.

Our work considers model merging in architectural descriptions. As in MDE, we also specify relations, automate composition and provide formal semantics for conformance checking and merging elements. The main difference is that where MDE approaches are defined within the context of the meta-object facility, we focussed on embedding these relations and compositions in a typical ADL and provide support for common architectural models such as C&C, infrastructure and deployment models. The latter brings model composition to software architecture.

AO. Aspect-orientation (AO) focusses on separating secondary or supporting functions (aspects) from the main functionality. The initial focus of AO was on programming languages, but this has later been extended to design and requirements.

Similar to AO, our work separates certain concerns (in models), and uses composition to integrate the models. Consequently, AO model composition approaches are related. Part of this related work overlaps with MDE and is discussed above, but aside from this there are related architectural approaches. A first category of related approaches focuses on separating concerns by adding behavior to architectural elements in ADLs (such as Aspectual ACME of Garcia et al. (2006), DAOP-ADL of Pinto et al. (2005), and the work of Grundy (2000)). We, on the contrary, separate concerns by using separate models. A second category of related approaches follows the

principles of multi-dimensional separation of concerns (MDSOC) by Tarr et al. (1999) on architecture (e.g. Kandé (2003); Baniassad et al. (2006)). This is somehow related, as models could correspond to hyperslices and composition to hypermodules. The main distinguishing factors of our approach are the explicit nature of relations, embedding relations and compositions into the AD, and our support for C&C, infrastructure and deployment models.

Alternative tool support. We briefly discuss possible alternatives to composition using state-of-the-art UML design tools.

Basic support for copying, pasting or cloning parts helps for the initial specification of integrated models, since elements must not be specified from scratch. However, the architect still needs to manually integrate the models and copy/pasting is little help to consistently change the AD.

Somewhat more advanced is support for element repositories. For example, in IBM's software architect the Project Explorer pane contains a list of all elements in a design project (IBM (2008)). Each element can be used in several diagrams, e.g. by dragging the element from the list onto a diagram. Other example tools with similar support are Visio and MagicDraw.

Element repositories are a useful addition, but do not replace composition. Dragging and dropping prevents re-specifying elements from scratch. Also, changes to an element are synced to all diagrams through the repositories. However, an architect still has to select the appropriate element from the repository for the integrated model. The selection must be reconsidered with each change, as a change might add or remove an element from the integrated model and these changes are not automatically synced. For example, links added between elements are not automatically synchronized between the models. Also, there is no explicit specification of which models are composed in an integrated model, so updates may be forgotten. An interesting idea is to combine element repositories with composition. This needs further investigation.

8. Conclusion

This paper investigates how automated composition of architectural models can help preventing problems associated with repetition in integrated models. The paper includes an empirical study of the problem, and presents an ADL based composition approach with formal semantics, and an empirical evaluation. The paper shows that composition of architectural models can be a powerful mechanism for software architects. An architect no longer specifies the integrated models directly, but instead specifies a composition that results in the same integrated model. Composition relieves the architect from the cumbersome task of specifying and maintaining repetition in integrated models. The cost is that the alternative AD will be larger in size.

The important lessons learned are:

- It is sometimes stated that composing heterogenous models is 'the real challenge', like by Nuseibeh et al. (2003). From the empirical study in this paper we learned that integrated C&C models and deployment models make up a substantial part of the AD. From our research it became clear that embedding this type of composition in ADs is already non-trivial and solves a real problem.

- Composition can only be used in practice when supported by languages and tools. The lack of standard languages and tools is a handicap for software architects.

- In this paper we add composition to a finished architectural document. But composition also influences the way architects design systems. For example, composition influences the way the AD is structured in models and views. This needs further investigation.

Interesting tracks for future research are broadening the empirical study to more ADs and more types of architectural models. Currently, we are looking to relations between state-charts and C&C models and their influence on the composition of state-chart. Another possibility for the future is adding an automated identification of relations. This would be a valuable addition to our approach.

Vitae

Nelis Boucké is a post-doctoral researcher at the Katholieke Universiteit Leuven since 2009. His PhD research was funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders and focussed on relations and compositions in the context of ADLs. During his PhD he also studied software architecture in general, aspect-oriented software development and multiagent systems. His current research interests focus mainly on software architecture, including architectural design, documentation and evaluation; languages and tools to support the day to day job of software architects; and software product line architectures. He has a strong interest to investigate these topics in a real-world setting and work towards practical solutions in cooperation with industry.

Danny Weyns is a post-doctoral researcher at the Katholieke Universiteit Leuven, funded by the Research Foundation Flanders. He received a PhD in Computer Science in 2006 from the Katholieke Universiteit Leuven for work on multiagent systems and software archtitecture. Danny's main research interests are in software architecture, self-adaptive systems, multiagent systems, and middleware for decentralized systems.

Tom Holvoet is a professor in the Department of Computer Science, Katholieke Universiteit Leuven. His research interests include software engineering of decentralized and multi-agent systems, coordination, software architecture, and autonomic computing. He received his PhD in computer science from the Katholieke Universiteit Leuven in 1997 on open distributed software development. Tom heads a research team with three post-doctoral and 6 PhD students within the DistriNet Lab.

Acknowledgment

Nelis is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Danny is supported by the Research Foundation Flanders (FWO Vlaanderen). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven. The authors thank Alexander Helleboogh and Dries Langsweirdt for the valuable discussions regarding the paper.

Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D., 2008. Differencing and merging of architectural views. Automated Software Engineering. 15 (1), 35–74.

Baniassad, E., Clements, P. C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B., 2006. Discovering early aspects. IEEE Softw. 23 (1), 61–70.

Barais, O., Klein, J., Baudry, B., Jackson, A., Clarke, S., Feb. 2008. Composing multi-view aspect models. pp. 43–52.

Bass, L., Clements, P., Kazman, R., 2003. Software Architectures in Practice. Addison.

Bendix, L., Emanuelsson, P., 2009. Requirements for practical model merge — an industrial perspective. In: MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin, Heidelberg, pp. 167–180.

Boucké, N., October 2008. Formal proofs of well-formedness and information-preservation properties and haskell source code. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW529.abs.html`.

Boucké, N., Holvoet, T., 2008. View composition in multiagent architectures. Special issue on Multiagent systems and software architecture, International Journal of Agent-Oriented Software Engineering (IJAOSE) 2 (2), 3–33.

Boucké, N., Weyns, D., Hilliard, R., Holvoet, T., Helleboogh, A., September 2008. Characterizing relations between architectural views 5292, 66–81.
URL `https://lirias.kuleuven.be/handle/123456789/198294`

Boucké, N., Weyns, D., Schelfthout, K., Holvoet, T., 2006. Applying the ATAM to an architecture for decentralized control of a transportation system. In: Quality of Software Architectures conference (QoSA). Vol. LNCS 4214.

Bzivin, J., S., B., Fabro, M. D. D., M., G., Jouault, F., Kolovos, D., I., K., Paige, R., 2006. A canonical scheme for model composition. In: Model Driven Architecture Foundations and Applications. Vol. 4066/2006. p. Model Management and Transformations.

Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A., Fernandes da Silva, L., 2009. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development. ACM, New York, NY, USA, pp. 149–160.

Clarke, S., Baniassad, E., 2005. Aspect-Oriented Analysis and Design. Addison-Wesley.

Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2003. Documenting Software Architectures, Views and Beyond. Addison Wesley.

Dashofy, E., van der Hoek, A., Taylor, R., 2005. A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology (TOSEM) 14 (2), 199–245.

Egyed, A., 2000. Heterogeneous view integration and its au-

tomation. Ph.D. thesis, Los Angeles, CA, USA, adviser-Barry William Boehm.

Egyed, A., Medvidovic, N., 2000. A formal approach to heterogeneous software modeling. In: FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering. Springer-Verlag, London, UK, pp. 178–192.

Fabro, M. D. D., Bézivin, J., Valduriez, P., 2006. Weaving models with eclipse amw plugin. Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany.

Fleurey, F., Baudry, B., France, R., Ghosh, S., 2008. A generic approach for automatic model composition. Springer-Verlag, Berlin, Heidelberg, pp. 7–15.

Garcia, A., Chavez, C., Batista, T., Sant'anna, C., Kulesza, U., Rashid, A., Lucena, C., 2006. On the modular representation of architectural aspects. In: Proc. of the European Workshop on Software Architecture.

Garlan, D., Monroe, R., Wile, D., 2000. ACME: Architectural description of component-based systems. In: Foundations of Component-Based Systems. Cambridge University Press.

Giese, H., Vilbig, A., June 2006. Separation of non-orthogonal concerns in software architecture and design. Software and Systems Modeling 5 (2), 136–169.

Grundy, J., December 2000. Multi-perspective specification, design and implementation of components using aspects. International Journal of Software Engineering and Knowledge Engineering 10 (6).

Grundy, J. C., Hosking, J. G., 2003. Softarch: Tool support for integrated software architecture development. International Journal of Software Engineering and Knowledge Engineering 13 (2), 125–151.

Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S., 2007. An algebraic view on the semantics of model composition. In: Akehurst, D. H., Vogel, R., Paige, R. F. (Eds.), ECMDA-FA. Vol. 4530 of Lecture Notes in Computer Science. Springer, pp. 99–113.

IBM, 2008. Rational Software Architect. IBM, `http://www-01.ibm.com/software/awdtools/architect/swarchitect/`.

ISO, July 2007. ISO/IEC 42010 Systems and Software Engineering – Architectural Description. ISO.

Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H., 2007. Model composition and feature interaction detection in product lines using critical pair analysis. In: International Conference on Model Driven Engineering Languages and Systems (MODELS).

Jeanneret, C., France, R., Baudry, B., 2008. A reference process for model composition. In: AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling. ACM, New York, NY, USA, pp. 1–6.

Kandé, M., 2003. A concern-oriented approach to software architecture. Ph.D. thesis, École Polytechnique Fédérale de Lausanne.

Katara, M., Katz, S., 2003. Architectural views of aspects. In: Proceedings International conference on Aspect-oriented software development. pp. 1–10.

Kolovos, D., Paige, R., Polack, F., 2006. Merging models with the epsilon merging language (eml). In: In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006. pp. 215–229. URL `http://dx.doi.org/10.1007/11880240\_16`

Landuyt, D. V., de beeck, S. O., Kemper, B., Truyen, E., Joosen, W., June 2008. Building a next-generation digital publishing platform using aosd. `http://distrinet.cs.kuleuven.be/projects/digitalpublishing/`.

Mahieu, T., Joosen, W., Van Landuyt, D., Grégoire, J., Buyens, K., Truyen, E., March 2007. System requirements on digital newspapers. CW Reports CW484, K.U.Leuven, Department of Computer Science.
URL `https://lirias.kuleuven.be/handle/123456789/156662`

Medvidovic, N., Gruenbacher, P., Egyed, A., Boehm, B. W., December 2003. Bridging models across the software lifecycle. Journal of Systems and Software 68 (3), 199–215.

Mehra, A., Grundy, J., Hosking, J., 2005. A generic approach to supporting diagram differencing and merging for collaborative design. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, New York, NY, USA, pp. 204–213.

Molesini, A., Garcia, A. F., Chavez, C. v. F. G., Batista, T. V., 2008. On the quantitative analysis of architecture stability in aspectual decompositions. In: WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). IEEE Computer Society, Washington, DC, USA, pp. 29–38.

Moriconi, M., Qian, X., 1994. Correctness and composition of software architectures. In: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering. ACM, New York, NY, USA, pp. 164–174.

Nejati, S., 2008. Behavioural model fusion. Ph.D. thesis, University of Toronto.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P., 2007. Matching and merging of statecharts specifications. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 54–64.

Nuseibeh, B., Kramer, J., Finkelstein, A., 2003. Viewpoints: meaningful relationships are difficult! In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 676–681.

Oliveira, K., Garcia, A., Whittle, J., 2008. On the quantitative assessment of class model compositions: An exploratory study. In: Workshop on Empirical Studies of Model-Driven Engineering.

Pinto, M., Fuentes, L., Troya, J. M., 2005. A dynamic component and aspect-oriented platform. Computing Journal 48 (4), 401–420.

Rubin, J., Chechik, M., Easterbrook, S. M., 2008. Declarative approach for model composition. In: MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering. ACM, New York, NY, USA, pp. 7–14.

Sabetzadeh, M., Nejati, S., Easterbrook, S., Chechik, M., 2006. A relationship-driven approach to view merging. SIGSOFT Softw. Eng. Notes 31 (6), 1–2.

Sabetzadeh, M., Nejati, S., Easterbrook, S., Chećhik, M., 2008. Global consistency checking of distributed models with tremer+. In: ICSE '08: Proceedings of the 30th international conference on Software engineering. ACM, New York, NY, USA, pp. 815–818.

Tarr, P., Ossher, H., Harrison, W., Sutton, S., 1999. N degrees of separation: Multi-dimensional separation of concerns. In: Int. Conf. on Software Engineering. pp. 107–119.

Van Landuyt, D., Grégoire, J., Michiels, S., Truyen, E., Joosen, W., October 2006. Architectural design of a digital publishing system. CW Reports CW465, Department of Computer Science, K.U.Leuven, Leuven, Belgium.
URL `https://lirias.kuleuven.be/handle/123456789/131455`

Weyns, D., Holvoet, T., 2008. Architectural design of a situated multiagent system for controlling automatic guided vehicles. International Journal on Agent Oriented Software Engineering 2 (1), 90–128.

Appendix A. Formal Underpinning

This appendix provides the details of the formal underpinning of the basic ADL, relations and composition. We discuss successively the naming conventions, helper functions, well-formedness rules, traceability and semantic preservation.

Appendix A.1. Naming Conventions

Below there is an overview of the abbreviations for C&C models. Since the name is shown in subscript, it is not shown in the tuples defining the naming conventions.

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

The abbreviations for infrastructure and deployment models:

$node_{name}$	Node
$compath_{name} = \langle nodes \rangle$	Communication path
$infra_{name} = \langle nodes, paths \rangle$	Infrastructure model
$dnode_{name} = \langle comps, cons, links \rangle$	Node
$dcompath_{name} = \langle nodes, links \rangle$	Communication path
$deployment_{name} = \langle nodes, paths \rangle$	Infrastructure model

The abbreviations for relations:

$iu = \langle ui1, ui2, newName \rangle$	Interface unification
$unif_{name} = \langle elem1, elem2, newName, ius \rangle$	Unification
$submodel_{name} = \langle target, submodel, ims \rangle$	Submodel
$alloc_{name} = \langle node, ccs \rangle$	Allocation

Appendix A.2. Helper Functions

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 translated to several functions for each of the types. Boucké (2008) provides the full specification of these functions.

$ccs : CCMODEL, SUB \rightarrow \mathbb{P}\ CC$: returns all components and connectors in the given C&C model or substructure; we use the abbreviated notation *ccmodel.ccs* and *sub.ccs*.

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

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

$allInts : CC, CCMODEL, \mathbb{P}CC, \mathbb{P}CCMODEL, DEPLOY, \mathbb{P}DEPLOY \rightarrow \mathbb{P}\ INT$: 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.

Appendix A.3. Well-formedness Rules

Well-formedness rules exclude invalid and redundant tuples from the formal definition of the models, relations and composition specifications.

Appendix A.3.1. Well-formedness of C&C models

$$\forall i_{name1}, i_{name2} \in INT : name1 = name2 \Rightarrow i_1 = i_2$$
$$\forall c_{name1}, c_{name2} \in CC : name1 = name2 \Rightarrow c_1 = c_2 \quad \text{(A.1)}$$
$$\ldots$$
$$\forall im \in IM : im.outer \neq im.inner \wedge im.outer.dir = im.inner.dir$$
$$\forall l \in LINK : (l.int1.dir = inout \wedge l.int2.dir = inout) \vee \quad \text{(A.2)}$$
$$(l.int1.dir = in \wedge l.int2.dir = out) \vee (l.int1.dir = out \wedge l.int2.dir = in)$$

$$\forall ccm \in CCMODEL : \forall l \in ccm.links : \{l.int1, l.int2\} \subseteq (ints\ ccm.comp \cup ints\ ccm.con) \quad \forall sub \in SUB, \forall l \in sub.links : \{l.int1, l.int2\} \subseteq (ints\ sub.comp \cup ints\ sub.con) \tag{A.3}$$

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

$$\forall ccm \in CCMODEL, \forall con \in ccm.con, \exists l \in struct.links : (l.int1 \in con.ints \Rightarrow \exists comp \in ccm.comp : l.int2 \in comp.ints) \wedge (l.int2 \in con.ints \Rightarrow \exists comp \in ccm.comp : l.int1 \in comp.ints) \tag{A.5}$$

Rule A.1 ensures that each element with a specific name is unique. The rule is only shown for interfaces and components. For connectors and C&C models, the definition is similar. Rule A.2 ensures that the directions of interfaces in interface mappings and links are compatible. Rules A.3 and A.4 make sure that links and interface mappings refer to interfaces of the correct C&C model 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 A.5 ensures that connectors can only be used between components. We only show the rule for C&C models, the definition for substructures is similar.

Appendix A.3.2. Well-formedness of Infrastructure and Deployment Models

The same uniqueness requirement as defined in rule A.1 applies on all elements in the infrastructure and deployment models having a name.

$$\forall compath \in COMPATH : compath.nodes \neq \emptyset \quad \forall dcompath \in COMPATH : dcompath.nodes \neq \emptyset \tag{A.6}$$

$$\forall infra \in INFRA, \forall compath \in infra.paths : compath.nodes \subseteq infra.nodes \quad \forall deploy \in INFRA, \forall dcompath \in deploy.paths : dcompath.nodes \subseteq deploy.nodes \tag{A.7}$$

$$\forall dnode \in DNODE : \forall l \in dnode.links : \{l.int1, l.int2\} \subseteq (ints\ dnode.comp \cup ints\ dnode.con) \tag{A.8}$$

$$\forall deploy \in DEPLOY, \forall dcompath \in DCOMPATH, \forall l \in dcompath.links, \exists dnode_1, dnode_2 \in deploy.nodes, \exists c_1 \in dnode_1.ccs, c_2 \in dnode_2.ccs : dnode_1, dnode_2 \subseteq dcompath.nodes \wedge l.int1 \in c_1.ints \wedge l.int2 \in c_2.ints \tag{A.9}$$

Rule A.6 states that each communication path should connect nodes. Rule A.7 states that the nodes in a communication path must be in the nodes of the respective infrastructure or deployment model. Rule A.8 states that the links in a dnode must be to components which are also in this node. Rule A.9 states that the links in a dcompath must be between components of the nodes connected by this path.

Appendix A.3.3. Well-formedness Rules for Relations

The same uniqueness requirement as defined in rule A.1 applies to the relations. Furthermore, the function allCC is also defined for relations.

$$\forall unif \in UNIF, \exists s_1, s_2 \in CCMODEL : (unif.elem1 \in s_1.comps \wedge unif.elem2 \in s_2.comps) \vee (unif.elem1 \in s_1.cons \wedge unif.elem2 \in s_2.cons) \tag{A.10}$$

$$\forall unif \in UNIF : (unif.elem1.sub = unif.elem2.sub) \vee unif.elem1.sub = none \vee unif.elem2.sub = none \tag{A.11}$$

$$\forall unif \in UNIF, \forall s_1, s_2 \in CCMODEL : unif.elem1 \in s_1.ccs \wedge unif.elem2 \in s_2.ccs \Rightarrow s_1 \neq s_2 \tag{A.12}$$

$$\forall unif \in UNIF, \forall iu \in unif.ius : (iu.ui1 \in unif.elem1.ints \wedge iu.ui2 \in unif.elem2.ints) \tag{A.13}$$

$$\forall submodel \in SUBMODEL : submodel.target \notin (allCC\ submodel.submodel) \tag{A.14}$$

$$\forall submodel \in SUBMODEL, \forall im \in submodel.ims, \exists cc \in submodel.submodel.ccs : im.outer \in submodel.target.ints \wedge im.inner \in cc.ints \tag{A.15}$$

Rule A.10 ensures that unifications between a component and a connector are not possible. Rule A.12 ensures that two unified elements are always from two different C&C models. Rule A.13 ensures that all interface unifications are between the interfaces of the unified elements. Rule A.11 ensures that the substructures are compatible. Rule A.14 ensures that the target element is not an element of the submodel. Rule A.15 ensures that interface mappings in relations are indeed between the related elements. There are no specific well-formedness rules for the internals of allocation links.

Appendix A.3.4. Well-formedness Rules for Composition Specifications

$$\forall spec \in COMPSPEC, \forall rel_1, rel_2 \in spec.inRels : rel_1, rel_2 \in SUBMODEL \cup UNIFICATION \wedge rel_1 \neq rel_2 \Rightarrow (allCC\ rel_1 \cap allCC\ rel_2 = \emptyset) \tag{A.16}$$

$$\forall spec \in COMPSPEC, \forall rel_1, rel_2 \in spec.inRels : rel_1, rel_2 \in ALLOC \wedge rel_1 \neq rel_2 \Rightarrow (allCC\ rel_1 \cap allCC\ rel_2 = \emptyset) \tag{A.17}$$

$$\forall spec \in COMPSPEC, \forall rel \in spec.inputRels : allCC\ rel \subseteq allCC\ spec.inputModels \wedge allInts\ rel \subseteq allInts\ spec.inputModels \tag{A.18}$$

Currently, no overlap between relations is allowed because we want to prevent interference between the relations. This could be relaxed in the future, but

we need to further investigate the interplay between the relations. Rule A.16 ensures that two relations between C&C models do not overlap. Rule A.17 ensures there is no overlap between allocation relations. Rule A.18 ensures that all relations are within the scope of the input models.

Appendix A.4. Traceability

We define four additional sets: inModels and inRels are the sets that correspond to the input models and relations for the composition function; outModel and traces represent the integrated model and the traces defined by the composition function.

The formal definition of traceability is captured as follows:

$$\forall model_i \in inModels : \langle outModel, model_i \rangle \in traces \tag{A.19}$$

$$\forall c_{out} \in allCC\ outModel, \exists model \in inModels, \exists c_{in} \in allCC\ model : \langle c_{out}, c_{in} \rangle \in traces \tag{A.20}$$

$$\forall node_{out} \in outModel.nodes, \exists model \in inModels, \exists node_{in} \in model.nodes : \langle node_{out}, node_{in} \rangle \in traces \tag{A.21}$$

$$\forall compath_{out} \in outModel.path, \exists model \in inModels, \exists compath_{in} \in model.path : \langle compath_{out}, compath_{in} \rangle \in traces \tag{A.22}$$

$$\forall c_{out} \in (allCC\ outModel), \forall i_{out} \in c_{out}.ints, \exists model \in inModels, \exists c_{in} \in (allCC\ model), \exists i_{in} \in c_{in}.ints : \langle c_{out}, c_{in} \rangle \in traces \wedge \langle i_{out}, i_{in} \rangle \in traces \tag{A.23}$$

$$\forall i_{out1}, i_{out2} \in (allInts\ outModel), \forall model \in inModels, \exists i_{in1}, i_{in2} \in (allInts\ model) : (\langle i_{out1}, i_{in1} \rangle \in traces \wedge \langle i_{out2}, i_{in2} \rangle \in traces \ \wedge connected\ i_{out1}\ i_{out2}) \Rightarrow connected\ i_{in1}\ i_{in2} \tag{A.24}$$

Rule A.19 states that there is a trace between the integrated model and each input model of a model composition. Rule A.20 states that for each component and connector in the integrated model, there exists a corresponding component or connector in the input models, identifiable through traces. Rule A.21 and rule A.22 state the same for nodes and compaths respectively. Rule A.23 states that for each interface of each component and connector in the integrated 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 integrated model, and the set of all elements with a name in the input models. Rule A.24 states that each connection in the integrated 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 Appendix A.2). Note that links are not directly traceable. Traces of links are implied by the connections.

Appendix A.5. Semantic Preservation

Appendix A.5.1. Model Consistency

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 integrated 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, nodes and communication paths. This is captured in the following formal statements:

$$\forall c_{out} \in allCC\ outModel, \forall model_1, model_2 \in inModels, \exists c_1 \in allCC\ model_1, \exists c_2 \in allCC\ model_2 : (\langle c_{out}, c_1 \rangle \in traces \wedge \langle c_{out}, c_2 \rangle \in traces \wedge c_1 \neq c_2) \Rightarrow model_1 \neq model_2 \tag{A.25}$$

$$\forall node_{out} \in outModel.nodes, \forall model_1, model_2 \in inModels, \exists node_1 \in model_1.nodes, \exists node_2 \in model_2.nodes : (\langle node_{out}, node_1 \rangle \in traces \wedge \langle node_{out}, node_2 \rangle \in traces \wedge node_1 \neq node_2) \Downarrow model_1 \neq model_2 \tag{A.26}$$

$$\forall compath_{out} \in outModel.path, \forall model_1, model_2 \in inModels, \exists compath_1 \in model_1.path, \exists compath_2 \in model_2.path : (\langle compath_{out}, compath_1 \rangle \in traces \wedge \langle compath_{out}, compath_2 \rangle \in traces \wedge node_1 \neq node_2) \Downarrow model_1 \neq model_2 \tag{A.27}$$

$$\forall c_{out} \in allCC\ outModel, \forall i_{out} \in c_{out}.ints, \exists model_1, model_2 \in inModels, \exists c_1 \in (allCC\ model_1), \exists c_2 \in (allCC\ model_2), \exists i_1 \in c_1.ints, \exists i_2 \in c_2.ints : \langle c_{out}, c_1 \rangle \in traces \wedge \langle c_{out}, c_2 \rangle \in traces \wedge \langle i_{out}, i_1 \rangle \in traces \wedge \langle i_{out}, i_2 \rangle \in traces \wedge i_1 \neq i_2 \Downarrow model_1 \neq model_2 \tag{A.28}$$

Rule A.25 states that for each component and connector in an integrated model, if there are two corresponding components or connectors in the input models, these components or connectors must be from

different input models. Rule A.26, rule A.27 and rule A.28 state the same for nodes, compaths and interfaces respectively.

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

$$\forall c_{in} \in inModels.ccs, \forall c_{insub} \in c_{in}.sub.css, \nexists c_{out} \in outModel.ccs : \langle c_{out}, c_{insub} \rangle \in traces \tag{A.29}$$

$$\forall c_{in} \in inModels.ccs, \forall c_{insub} \in c_{in}.sub.css, \exists c_{out} \in outModel.ccs, \exists c_{outsub} \in c_{out}.sub.ccs : \langle c_{outsub}, c_{insub} \rangle \in traces) \Rightarrow \langle c_{out}, c_{in} \rangle \in traces \tag{A.30}$$

Rule A.29 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 integrated model. Rule A.30 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 ensures that the element is part of the same substructure in the integrated model.

Appendix A.5.2. Model Completeness

$$\forall model_{in} \in inModels, \forall c_{in} \in allCC\ model_{in}, \exists! c_{out} \in allCC\ outModel : \langle c_{out}, c_{in} \rangle \in traces \tag{A.31}$$

$$\forall model_{in} \in inModels, \forall node_{in} \in model_{in}.nodes, \exists! node_{out} \in outModel.node : \langle node_{out}, node_{in} \rangle \in traces \tag{A.32}$$

$$\forall model_{in} \in inModels, \forall compath_{in} \in model_{in}.path, \exists! compath_{out} \in outModel.path : \langle compath_{out}, compath_{in} \rangle \in traces \tag{A.33}$$

$$\forall model \in inModels, \forall c_{in} \in allCC\ model, \forall i_{in} \in ccInts\ c_{in}, \exists! c_{out} \in allCC\ outModel, \exists! i_{out} \in ccInts\ c_{out} : \langle c_{out}, c_{in} \rangle \in traces \wedge \langle i_{out}, i_{in} \rangle \in traces \tag{A.34}$$

$$\forall model \in inModels, \forall c_{in} \in (allCC\ model), \exists! c_{out} \in (allCC\ outModel) : \langle c_{out}, c_{in} \rangle \in traces \wedge (c_{in}.sub = notspecified \vee (\forall c_{insub} \in allCC\ c_{in}, \exists! c_{outsub} \in allCC\ c_{out} : \langle c_{outsub}, c_{insub} \rangle \in traces)) \tag{A.35}$$

$$\forall model \in inModels, \forall i_{in1}, i_{in2} \in (allInts\ model), \exists! i_{out1}, i_{out2} \in (allInts\ outModel) : \langle i_{out1}, i_{in1} \rangle \in traces \wedge \langle i_{out2}, i_{in2} \rangle \in traces \wedge connected\ i_{in1}\ i_{in2} \Downarrow connected\ i_{out1}\ i_{out2} \tag{A.36}$$

Rule A.31 states that for each component and connector in the input models, there exist a unique counterpart in the integrated model, identifiable through traces. Rules A.32 and A.33 state the same for nodes and communication paths. Rule A.34 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 integrated model. Rule A.35 states that for each component and connector in the input models, the corresponding component or connector in the integrated model has a corresponding substructure. Rule A.36 states that for each connection between interfaces in the input models, there exists a connection between the corresponding interfaces in the integrated 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 integrated model with a name.

Appendix A.5.3. Relational Completeness

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

$$\forall unif_i \in inRel \cap UNIF, \exists! cc_{out} \in (allCC\ outModel), \exists t_1, t_2 \in traces : \begin{pmatrix} t_1 = \langle cc_{out}, unif_i.elem1 \rangle \wedge t_2 = \langle cc_{out}, unif_i.elem2 \rangle \wedge \\ (\nexists t_3 \in trace : t_3.out = cc_{out} \wedge t_3 \neq t_1 \wedge t_3 \neq t_2) \end{pmatrix} \wedge \begin{pmatrix} \forall iu_j \in unif_i.ius, \exists! i_{out} \in cc_{out}.ints, \exists t_1, t_2 \in traces : \\ t_1 = \langle i_{out}, iu_j.ui1 \rangle \wedge t_2 = \langle i_{out}, iu_j.ui2 \rangle \wedge \\ (\nexists t_3 \in trace : t_3.out = i_{out} \wedge t_3 \neq t_1 \wedge t_3 \neq t_2) \end{pmatrix} \wedge \begin{pmatrix} \forall i_{in} \in (unif_i.elem1.ints \cup unif_i.elem2.ints) / allInts\ unif_i.ius, \\ \exists i_k \in cc_{out}.ints : \langle i_k, i_{in} \rangle \in traces \end{pmatrix} \tag{A.37}$$

$$\forall submodel_i \in inRel \cap SUBMODEL, \exists! cc_{out} \in (allCC\ outModel) : \langle cc_{out}, submodel_i.target \rangle \in traces \wedge (\forall cc \in submodel_i.submodel.ccs, \exists! cc_{subout} \in cc_{out}.sub : \langle cc_{subout}, cc \rangle \in traces) \tag{A.38}$$

$$\forall alloc \in inRel \cap ALLOC, \exists! node_{out} \in outModel.nodes : \langle node_{out}, alloc.node \rangle \in traces \wedge (\forall cc_{in} \in alloc.ccs, \exists! cc_{out} \in node_{out}.ccs : \langle cc_{out}, cc_{in} \rangle \in traces) \tag{A.39}$$

$$\forall model \in inModels, \forall cc_{in} \in model.ccs / allCC\ inRel, \exists! cc_{out} \in outStr.ccs, \exists t_{c1} \in traces : t_{c1} = \langle cc_{out}, cc_{in} \rangle \wedge (\nexists t_{c2} \in traces : t_{c2}.out = cc_{out} \wedge t_{c2} \neq t_{c1}) \wedge \begin{pmatrix} \forall i_{in} \in cc_{in}.ints, \exists! i_{out} \in cc_{out.ints}, t_{i1} \in traces : t_{i1} = \langle i_{in}, i_{out} \rangle \wedge \\ (\nexists t_{i2} \in traces : t_{i2}.out = i_{out} \wedge t_{i2} \neq t_{i1}) \end{pmatrix} \tag{A.40}$$

Rule A.37 defines what information is preserved from a unification. The first part of the rule (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.

Rules A.38 and A.39 define the semantics of submodel and allocation respectively. Finally, rule A.40 states that all components and connectors (and their interfaces) that are not part of a relation in the input models must be uniquely traceable to an element in one of the integrated models. This last rule partly overlaps with the definition of completeness, but with the additional requirement that there is a *unique* trace.