Composing architectural crosscutting structures in xADL

Nelis Boucke, Alessandro Garcia and Tom Holvoet

Report CW474, January 2007

Composing architectural crosscutting structures in xADL

Nelis Boucke, Alessandro Garcia and Tom Holvoet

Report CW474, January 2007

Department of Computer Science, K.U.Leuven

Abstract

Designing a software architecture is about defining and composing highlevel design structures. Whereas describing several structures is is fairly well supported, both non-aspect-oriented and aspect-oriented Architectural Description Languages (ADLs) fall short when it comes to documenting relations or compositions between architecturally-relevant crosscutting structures. This makes in turn separation of important concerns in the architecture hard, thereby increasing maintenance overhead and reducing reuse capabilities. This paper identifies and analyzes examples of crosscutting structures in an architecture for an industrial Automatic Guided Vehicle Transportation System (AGVTS). This analysis allowed us to determine and introduce an initial set of structural composition operators into xADL, namely substructure, mapping, and unification. The operators' feasibility have been assessed while refactoring the existing AGVTS architecture. Based on a real maintenance scenario, we also investigate to what extent these explicit compositions led (or not) to enhanced architectural changeability for evolving the distribution strategy.

Keywords : structural composition, ADL, xADL
CR Subject Classification : D.2

Composing Architectural Crosscutting Structures in xADL

Nelis Boucké[1], Alessandro Garcia[2], Tom Holvoet[1]

[1] Distrinet, KULeuven, {nelis.boucke,tom.holvoet}@cs.kuleuven.be
[2] Computing Department, Lancaster University, garciaa@comp.lancs.ac.uk

Abstract. Designing a software architecture is about defining and composing high-level design structures. Whereas describing several structures is is fairly well supported, both non-aspect-oriented and aspect-oriented Architectural Description Languages (ADLs) fall short when it comes to documenting *relations or compositions* between architecturally-relevant crosscutting structures. This makes in turn separation of important concerns in the architecture hard, thereby increasing maintenance overhead and reducing reuse capabilities. This paper identifies and analyzes examples of crosscutting structures in an architecture for an industrial Automatic Guided Vehicle Transportation System (AGVTS). This analysis allowed us to determine and introduce an initial set of structural composition operators into xADL, namely *substructure*, *mapping*, and *unification*. The operators' feasibility have been assessed while refactoring the existing AGVTS architecture. Based on a real maintenance scenario, we also investigate to what extent these explicit compositions led (or not) to enhanced architectural changeability for evolving the distribution strategy.

1 Introduction

Software architectures shift the focus from lines-of-code to coarser-grained software elements and their overall interconnection structure [32]. The architecture of a software system is commonly referred as the fundamental design structure or structures, which comprise software elements, the externally visible properties of those elements, and the relationships among them [4]. This implies that the core issue in architectural design is to define and relate (compose) the high-level design structures that are relevant to key stakeholders' concerns. As a result, over the last decades several Architectural Description Languages (ADLs) have been proposed to model such architecturally-relevant structures [32]. Two notable examples are xADL [14] and ACME [24], both modern general-purpose languages with tool support in the Eclipse development environment [15].

With the rise of Aspect-Orientated Software Development (AOSD [20]), there is a growing awareness that certain concerns have a structural and behavioral crosscutting impact on architectural design [2, 3]. A crosscutting concern in an ADL description is a concern that cannot be effectively modularized using the abstractions of that ADL. Not separating important concerns leads to increased maintenance overhead, reduced reuse capabilities, and generally results in architectural erosion over the system lifetime [5]. New aspect-oriented (AO) ADLs concentrate on defining aspectual components and innovative aspectual composition operators to add or change *behavior* on component interfaces [5, 35, 37]. In another words, they tend to mimic AO composition mechanisms only supported by the family of AspectJ [16]-like programming languages [28]. Some representative examples of this category of ADLs are AspectualACME [22], DAOP-ADL [37], and PRISMA [35]. However, empirical knowledge, based on industrial-strength case studies, about the interplay of crosscutting and ADL-based *structural decomposition* is lacking. It is generally accepted that an architectural description exists of several separated structures (referred to as views in [12, 26]) and that each of structure describes the system from the perspective of a particular concern. Each structure is a set of components, connectors, associated interfaces and the relations between them.

In this context, the problem we ran into when defining an architecture was that both non-AO and AO ADLs and associated tools are limited to support the description of architectural structures. However, there is little support to modularly describe *relations or compositions* between different structures, especially crosscutting structures. Yet, the concerns described in different structures are not independent form each other, and an architect must be able to explicitly define their composition. Explicit relations or advanced compositions are a major challenge for separating concerns where we believe that aspect-orientation can contribute to architectural descriptions. This is analogous with extending UML with model composition semantics for Theme/UML [11] or enhancing the Java language with structural composition mechanisms [28] like the ones supported by the Hyper/J programming language [40].

This paper describes a case study where we have identified recurring examples of crosscutting architecture structures. The investigation was rooted at the review of a significant excerpt of a real large-scale application, a multi-agent architecture for a industrial Automatic Guided Vehicle Transportation System (AGVTS). Basically, we have studied to what extent the xADL language supports modular description of such crosscutting structures in the AGVTS architecture. Based on this analysis, we determine and introduce an initial set of structural composition operators in xADL, namely *substructure*, *mapping*, and *unification*. We specifically choose to use xADL because it provides a rigourously defined language that can be extended easily. This is in contrast with our early explorations in [6, 8], providing only an informal definition of a similar type of composition. To assess their feasibility, we use the operators to refactor and evolve the AGVTS architecture in order to make it explicit key architectural structure concerns and their relations.

Overview: The remainder of this paper is structured as follows. Section 2 introduces ADLs and points out the need for relations between structures. Section 3 introduces our case study and illustrates the lack of relations in a more concrete context. In section 4 we introduce the composition operators and review an excerpt of the AGVTS in the context of this operators. Section 5 describes related work. Finally, we conclude in section 6.

2 Architectural description languages

This section introduces ADLs, and more specifically xADL, which will be used in this paper. This section also makes some initial reflections on the need for relations between structures. In the past, several ADLs have been proposed to model architectures, either to model a particular application domain or as general-purpose architectural languages [32]. The ADLs ACME and xADL are of particular interest because both languages provide a simple set of general purpose constructs to model architecture: components, connectors, interfaces and the relations between them. Recently, both ADLs receive more attention again because they provide a valuable alternative for the one-size-fits-all UML language, who provides less focus for architectural modeling with its many constituent notations and arbitrary extensions [31]. Additionally, the tool support for both languages has been considerably extended and integrated with the Eclipse platform. For instance, Archstudio 4.0 is a tool for xADL that has largely improved with its update in November 2006 [27].

2.1 Basic Elements of xADL

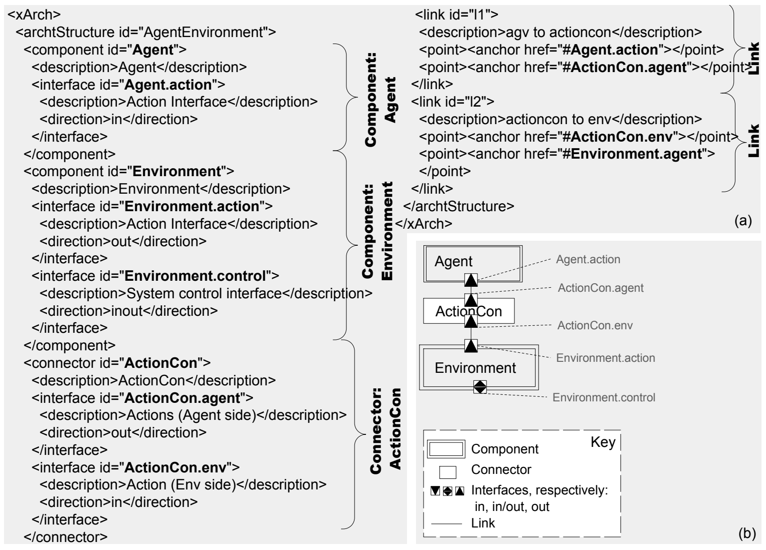
This paper relies on xADL to investigate the nature of certain crosscutting architecture structures and their compositions. We have chosen this language, because the goal of xADL is to provide a reusable core language and tool set that can be easily extended. xADL is an architectural description language built on top of XML. To allow straightforward extensions, the xADL language definition is specified in XSLT [1] language. Also, the ACME ADL uses the same basic elements and has the same structural properties as xADL. An architectural description in xADL is built around the following basic elements [14].

– **Components**: Components are the loci of computation in the architecture. Components have a unique identifier and a textual description, along with a set of interfaces.
– **Connectors**: Connectors are the loci of communication in the architecture design. Similar to components, connectors also have only a unique identifier, a textual description, and a set of interfaces.
– **Interfaces**: Interfaces are components' and connectors' portals to the outside world. The "interface" in xADL are also known as "ports" for components and "roles" for connectors in the ACME ADL. Interfaces have a unique identifier, a textual description, and a direction. The direction provides an indicator of whether the interface is provided, required, or both.
– **Links**: Links are connections between interfaces, defining the topology of the architecture.

xADL supports a distinction between types, static descriptions and dynamic descriptions. It encompasses several architectural structures for static descriptions, and instance diagrams for dynamic descriptions. Here we focus on the structural part of a software architecture. Concretely, an architectural structure contains a set of components, connectors, interfaces and links. A single architectural description in xADL can contain several architectural structures. The concept of architectural structure in xADL is strongly related to the concept of module views [12]. In this context, components and connectors refer to static entities, not run-time entities as in [4]. Figure 1 shows an example architectural description containing two components, `Agent` and `Environment`, one connector, `ActionCon`, and the several links to attach the connector to the components. In principle, the identifier of an interface can be any unique string. We follow the convention that the identifier uses the element name dot interface name to make the description more readable, like `Agent.action`. References are represented by the href property, containing a unique identifier or an element after the sharp symbol.

The specification of an architecture is not done by writing XML documents by hand, but by using the ArchStudio tool. ArchStudio contains both a visual editor, that allows to graphically manipulate the architecture (notation as in to fig. 1(b)) and a specification editor who represents the architecture in the form of a tree. The remainder of the paper uses graphical notations as much as possible for readability and understandability purposes, but XML descriptions are inevitable to provide details for composition specifications.

Special construct: subarchitecture xADL has an additional construct we will explain because one of our composition operators is inspired by it, called *subarchitecture*. Subarchitecture is a relation between a component or connector *type* and an architectural structure, defining that the architectural structure forms a subarchitecture of the component or connector type. To understand subarchitecture we have to briefly explain types in xADL.



```
<xArch>
 <archtStructure id="AgentEnvironment">
  <component id="Agent">
   <description>Agent</description>
   <interface id="Agent.action">
    <description>Action Interface</description>
    <direction>in</direction>
   </interface>
  </component>
  <component id="Environment">
   <description>Environment</description>
   <interface id="Environment.action">
    <description>Action Interface</description>
    <direction>out</direction>
   </interface>
   <interface id="Environment.control">
    <description>System control interface</description>
    <direction>inout</direction>
   </interface>
  </component>
  <connector id="ActionCon">
   <description>ActionCon</description>
   <interface id="ActionCon.agent">
    <description>Actions (Agent side)</description>
    <direction>out</direction>
   </interface>
   <interface id="ActionCon.env">
    <description>Action (Env side)</description>
    <direction>in</direction>
   </interface>
  </connector>
  <link id="l1">
   <description>agv to actioncon</description>
   <point><anchor href="#Agent.action"></point>
   <point><anchor href="#ActionCon.agent"></point>
  </link>
  <link id="l2">
   <description>actioncon to env</description>
   <point><anchor href="#ActionCon.env"></point>
   <point><anchor href="#Environment.agent">
   </point>
  </link>
 </archtStructure>
</xArch>
```

Fig. 1. Example xADL specification, showing the `Agent` and `Environment` component, connected with the `ActionCon` connector. We used the same abbreviated notation as the xADL authors used in [14] to improve readability of the specification. We will only once include a key, the other figures use the same graphical notation.

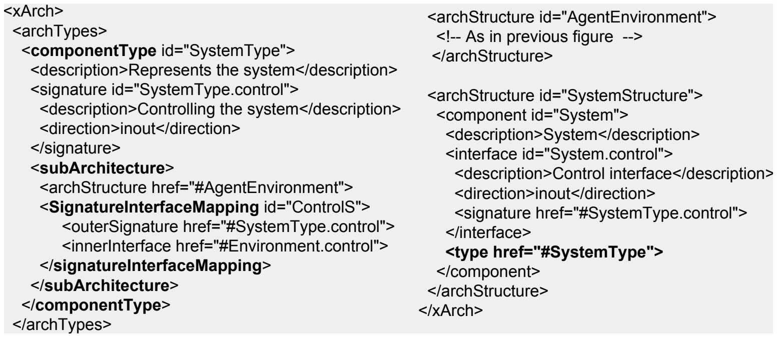
Types are grouped in the archTypes section of an xADL document, specifying types for components, connectors and interfaces. Component and connector types are defined as a set of signatures. Signature are the least intuitive concepts of xADL. A signature prescribes what interface a component must have to be of the type.

For example, consider the an example component type `SystemType` in fig. 2, having one signature called `SystemType.control`. The `SystemStructure` architectural structure uses the `SystemType` to construct the `System` component, and provides a link to show that the `System.control` interface covers the `SystemType.control` signature. The subarchitecture construct is contained within the type, e.g. in the `SystemType` component in fig. 2. Note that there appears a (indirect) relation between two architectural structures. E.g. SystemStructure is related with AgentEnvironment through its `System` component of type `SystemType`.

2.2 Motivation for structural relations in xADL

In this section we point out the need for structural relations in an ADL like xADL. The next section places this motivation in a more concrete context, based on an industrial case study.

Following the recommended practice for architectural description [26], an architect will identify several concerns an describe these concerns in different structures. At first glance, xADL seems to support this separation of concerns because it allows to describe several architectural structures. A second glance reveals that such separation is difficult, since there is limited support to *relate or compose* different structures. From our experience, the subarchitecture construct alone does not suffice in most circumstances, for instance because



```
<xArch>
 <archTypes>
  <componentType id="SystemType">
   <description>Represents the system</description>
   <signature id="SystemType.control">
    <description>Controlling the system</description>
    <direction>inout</direction>
   </signature>
   <subArchitecture>
    <archStructure href="#AgentEnvironment">
    <SignatureInterfaceMapping id="ControlS">
       <outerSignature href="#SystemType.control">
       <innerInterface href="#Environment.control">
    </signatureInterfaceMapping>
   </subArchitecture>
  </componentType>
 </archTypes>
```

```
 <archStructure id="AgentEnvironment">
  <!-- As in previous figure  -->
 </archStructure>

 <archStructure id="SystemStructure">
  <component id="System">
   <description>System</description>
   <interface id="System.control">
    <description>Control interface</description>
    <direction>inout</direction>
    <signature href="#SystemType.control">
   </interface>
   <type href="#SystemType">
  </component>
 </archStructure>
</xArch>
```

Fig. 2. Example with subarchitecture construct.

it is not possible to create a link between architectural elements in different structures or subarchitectures.

In this context, the architect is left with limited choices. One extreme is to describe the architecture using a single structure and completely give up the separation of concerns. This may be feasible for small systems, but for larger industrial systems this will fast become cumbersome. In the other extreme, the architect could keep a very strict separation of concerns, and rely on implicit relations between structures (e.g. names of components or a textual description of the architecture). But implicit relations make changing the architecture very hard, since there is no clear way to know the implication of a specific change. In practice, most architects will be obligated to give up a strict separation between concerns, concretely illustrated in our case study in the next section. They will still use several structures, but mingle several concerns in every structure to allow specifying the relations between them.

3 Case study: Automatic Guided Vehicle Transportation System

This section starts with describing our case study, an Automatic Guided Vehicle System (AGVTS) (in section 3.1). This application is a real industrial system, and is mainly based on a multi agent architecture. Section 3.2 exploits an excerpt from the existing architectural description for the AGVTS in order to illustrate: (i) the problem of concerns that crosscut structures, and (ii) the need for explicit structural relations (more concretely illustrating section 2.2).

3.1 Automatic Guided Vehicle Transport System

An Automatic Guided Vehicle Transport System (AGVTS) is a fully automated industrial system that uses multiple AGVs to transport loads in a warehouse or production plant. An AGV is an unmanned, battery powered transportation vehicle. The AGVTS has to interface with Ensor, a software system running on the computer of each AGV vehicle that steers the vehicle based on high-level commands given by the AGVTS (like move, turn). The main functional requirements for this system are: (1) allocating transportation tasks to individual AGVs; (2) performing those tasks; (3) preventing conflicts between AGVs on crossroads; and (4) charging the batteries of AGVs before they are drained. Transportation tasks are

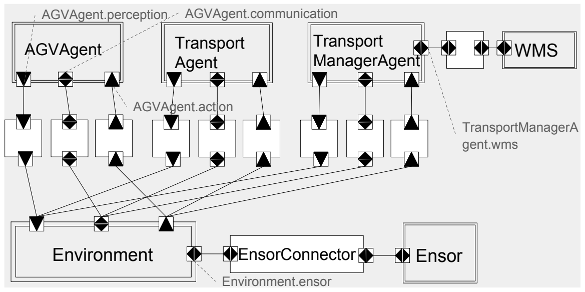


Fig. 3. Decomposition in agents and environment for AGVTS.

coming from a Warehouse Management System (WMS), a software system to manage products in an industrial warehouse.

Multi agent architecture. In a joint research project between Distrinet and Egemin [17], a producer of automated logistic systems, the AGVTS was built with a multi agent architecture. This architecture answers to new market demands, asking for flexible, open an robust systems that can handle dynamic operating conditions in an autonomous way. These qualities are ascribed to multi agent architectures [9]. The multi agent domain is of particular interest because it entails complex software architectures with rich categories of crosscutting concerns [30, 23]. Such architectures typically involve both general crosscutting concerns, such as distribution and security, and domain-specific concerns for multi-agent architectures such as coordination and decomposition in agents and environment.

Using a multi agent architecture means structuring the system in several autonomous entities (agent components) connected to an environment component. The agents and environment are built according to the reference architecture for multi agent systems [41]. An agent component operate autonomous, but coordinate with each other to achieve the system goals. Each agent has three interfaces: (1) perception: to observe the environment; (2) action: to influence the environment; (3) communication: to communicate with other agents by sending messages through the environment. The environment is a software component, shielding the agents from technical details and providing abstractions to view and manipulate the real world [42]. Concretely, the environment provides the interfaces for perception, action and communication.

Concerns. Based on our experience with building an architecture for the AGVTS, we identified three important concerns for the software architecture. The three concerns are decomposition in agents/environment, coordination and distribution.

The agents/environment concern covers the decomposition into different agent types and an environment. Figure 3 describes the agents and environment for the AGVTS. The `AGVAgent` is responsible for managing the tasks of an AGV. The `TransportAgent` must search an appropriate AGV to perform a new transport and do the follow up of each transport. Finally, the `TransportManagerAgent` is in charge of communicating with WMS and managing (create, remove, etc.) all `TransportAgent`s. The environment abstracts away the agents from several details, including the translation of high-level commands from the `AGVAgent` to `Ensor`.

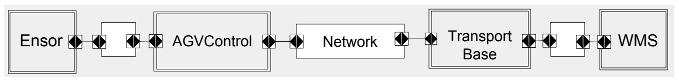


Fig. 4. Example subsystems for distribution.

For the coordination concerns, we focus on assigning transport tasks to AGVs. We use a standardized coordination protocol called Contract Net (CNET [21]), based on a marked mechanism. In CNET, an initiator calls for proposals and participants offer proposals to perform the task. When the initiator has received the proposals from all participants, it evaluates the proposals and assigns the task to the participant with the best offer. In case of the AGVTS, the `TransportAgent` plays the role of initiator, and the `AGVAgent` of participant. Proposals are based on the distance of an AGV vehicle to the start of the transport task.

The distribution concern covers the decomposition into sub systems to allow easy deployment on a physical infrastructure and any additional support needed for this, like remoting or synchronization software. For the AGVTS, the physical infrastructure exists of a industrial computer on each AGV vehicle and one or several server systems. Figure 4 shows an example decomposition in an `AGVControl` subsystem and a `TransportBase` subsystem, to be deployed on the AGV vehicle and a server respectively. In this example, the `AGVAgent` would be allocated on the `AGVControl` system, the `TransportAgent` and `TransportManagerAgent` on the `TransportBase`.

3.2 Crosscutting Structures in Existing Architectural Description

The existing architectural description for the AGVTS was built according to the guidelines in [12], but without the idea of aspect orientation in mind. The complete description can be found in [7]. Here, we converted the description to xADL and we only focus on a part of the architecture.

Agents/environment crosscuts with distribution. Figure 5 contains the first view of the existing architectural description. The remainder of the architecture is built on top of this decomposition. Careful analysis reveals that this decomposition mixes up at least two of the concerns: agents/environment and distribution.

The agent/environment concerns is represented by the agent types (`AGVAgent`, `TransportAgent`, `TransportManagerAgent`). The distribution concern is represented by the subsystems (`AGVControl` and `TransportBase`), who will be deployed on an AGV vehicle and a server respectively. The environment component itself is not present anymore, but split up in several local environments to allow matching the structure of the agent/environment and the subsystems for distribution.

This is a typical example of a structure intermingling several concerns. Architects are often obligated to used such intermingling. In the example, the architect of the AGVTS can describe the structure for the AGVTS separately using xADLs concepts of several architectural structures, but can not describe the relationships. Not separating important concerns and explicitly define the relations can have far reaching consequences. An architect who wants to change a concern must search within all structures, with few guidelines where to search. Because the relations between concerns are implicit, changing one concern in one

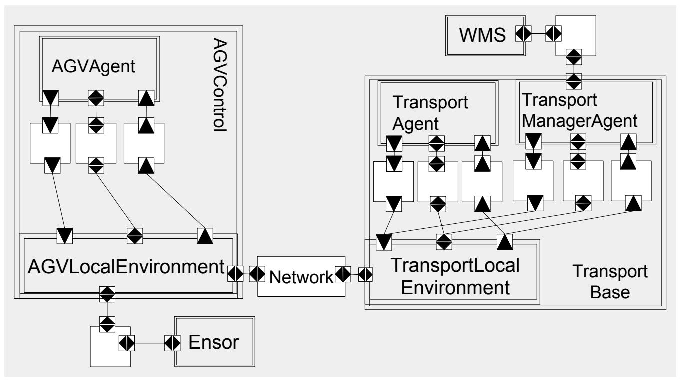


Fig. 5. First decomposition view of architectural documentation.

structure may have unforseen effects on other structures. As such, the whole architecture must be reexamined and possible adapted to change something to a single concern.

For example, distribution is not separated but the requirements changed during the project. The company wanted a decentralized architecture, with the agents spread over several computer system. During the project, the company realized that stepwise adoption would be easier by first deploying the multi agent architecture in a centralized way, with all agents on a single computer system, and later migrate to decentralized deployment. This allowed to initially better control over the system and interaction between the agents. Everything could stay the same, except for distribution. But because of structural crosscutting, we had to redefine the complete architecture and change nearly every individual structure of the architectural description. From our experience, this is a typical scenario in realistic iterative, incremental development processes.

Looking back to the definition of the agents/environment structure (fig. 3) and the distribution structure (fig. 4), one can see that relating this two structures to obtain fig. 5 is not trivial. We identify three types of related components between these two concerns: (1) The same components appear in different structures (`Ensor`, `WMS`), a simple type of overlap. (2) Some components must become subcomponents of other components, e.g. `AGVAgent` must become subcomponent of `AGVControl`. This is also a kind of overlap, but more complex the previous type of relation. (3) There is a structural mismatch between some components, `Environment` component does not match with the decomposition into the `AGVControl` and `TransportBase` subsystem, solved by splitting the `Environment` component. This is a first indication of relation types, we will come back to this while describing the composition operators in sec. 4.1.

Coordination crosscuts agents/environment. Similar problems occur for other concerns. As an illustration, we made an annotated table of content (fig. 6) for the architectural description of the AGVTS. Design model names marked in bold include fragments of the agent/environment concern, underlined names include fragments of the coordination concern. Design models that are both bold and underlined intermingle the definition of coordination with agents, illustrating that the two concerns crosscut each other in the design models. This illustrates the broadly scoped influence of coordination on the architectural structures.

- Static views (structures)
  1. Layered: The ATS (AGVs transportation system)
  2. Decomposition: **The ATS, Transport base, AGVControl, Local environment**, Transport agent, AGV agent, Decisions
  3. Uses: **Transport Base, AGV Control system**
  4. Generalization: **Agents**
- Runtime views (instances)
  1. Shared date: **Agent, Local virtual environment, Protocol description**
  2. Process views: **Move action, Sending-Receive action, Background processes**
- Mixed views: ObjectPlaces middleware, collision avoidance

Fig. 6. List of design models made for the AGV transportation system. Model names in bold include the agent/environment concern, underlined model names include the coordination concern.

4 Composing crosscutting structures

This section illustrates how to compose crosscutting structures in xADL. To do this, we first define three composition operators as an extension to xADL architectural descriptions in section 4.1. Next, we apply this composition operators in section 4.2 to refactor an excerpt of the AGVTS architecture covering the three concerns introduced earlier in this paper, agents/environment, coordination and distribution.

4.1 Composition Operators for Architectural Structures

Firstly, we add a new element to the xADL description called architectural composition (`<archComposition>` in the XML specification), describing a concrete composition of several structures. Next, we introduce three composition rules.

- **Unification**: Unifies elements (components or connectors) from different structures with each other, i.e. declares that the elements are exactly the same element. The unified elements must either have exactly the same interfaces, or the architect must define the corresponding interfaces.
- **Mapping**: Maps individual or groups of elements (called subjects) from one architectural structure on a single element of another architectural structure (called target). The subjects then become subelements of the target element. The join points used in the mapping rule are the interfaces, the architect must define corresponding interfaces between the target and the subject.
- **Substructure**: Defines that a specific architectural structure (referred to as inner structure) describes a substructure for an architectural component (referred to as outer component) of another architectural structure. The join points used in the mapping rule are again the interfaces, the architect must define corresponding interfaces between the outer component and interfaces in the inner structure.

Each of the composition operators has been introduced for a specific purpose, based on observations of possible relations in the AGVTS (near the end of section 3.2). The purpose of unification is to describe overlap between different structures. Mapping has the purpose to solve more difficult kind of overlap, i.e. that some components (but not all of them) of one structure are actually subcomponents of another structure. The purpose of substructure is to resolve structural mismatches, some elements of the original structures can be split up

Structures	**Purpose**	**Concern**
AgentEnvironment	Describes the agent types, environment and associated relations.	agent/environment
CNET	Describes the structure for the CNET protocol.	coordination
CNETConnectorInternal	Describes a substructure of the CNETConnector.	auxiliary, to compose AgentEnvironment and CNET.
CentralizedSubSystem	Subsystems needed for centralized deployment.	distribution (centralized)
EnsorConnectorInternal	Describes a substructure of the EnsorConnector.	auxiliary, to compose Centralize-Subsystem with previous.
DecentralizedSubSystem	Subsystems needed for decentralized deployment.	distribution (decentralized)
EnvironmentInternal	Describes a substructure of the Environment component.	auxiliary, to compose DecentralizedSubSystem with previous.

Fig. 7. The different structures made for the AGVTS.

until the architect can describe a useful relation between them. The details of the operators are illustrated in the next section, together with an example.

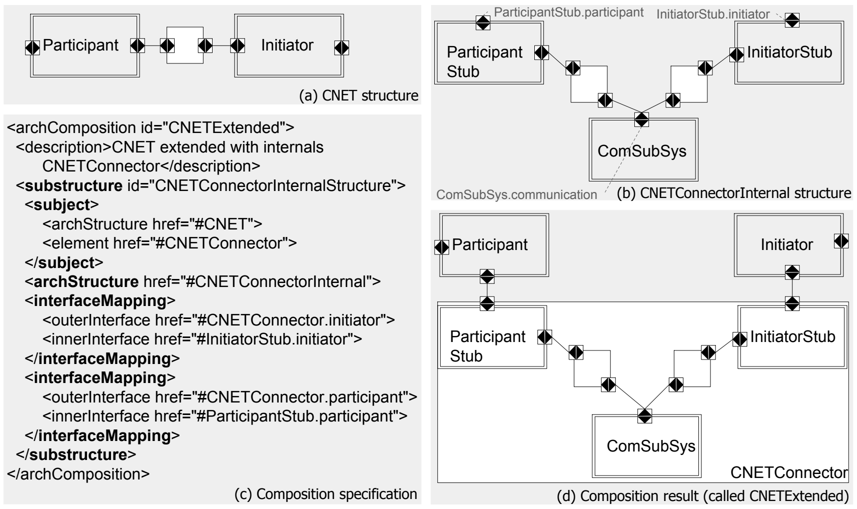
Notice that there are several important differences between the existing subarchitecture construct (described in section 2.1) and the substructure introduced here. Firstly, subarchitecture is a relation between a *type* and a structure. Substructure is a relation between two structures, resulting in a more direct relation between structures. Secondly, substructure is not described in the component type, as for subarchitecture, but in a separate composition specification section. This has the advantage that substructures can be varied depending on the context of the target composition.

4.2 Example of Composing Structures in AGVTS

Figure 7 provides an overview of the structures we have described for the AGVTS, with a short description of the purpose and the associated concern. In step 1 we start with describing the agent/environment and coordination concern. Step 2 adds the internal structure of the CNETConnector and describes the concrete composition. The next steps are to add distribution to the system, which is in turn based on two alternatives (as discussed before): centralized deployment (step C3 and C4) and decentralized deployment (step D3 and D4).

Step 1: Agents/environment and coordination. The agent/environment concern has already been explained in detail in section 3.2 and fig. 3. The coordination concern is represented by the CNET structure fig. 8(a). This structure shows the high-level components for the Contract Net protocol, the Initiator and Participant components. Both components have two interfaces: an interface to communicate with the other partner (connected by the CNETConnector) and an interface to the internals of the agent for getting state and synchronizing with the internal behavior of the agents. The internal architecture of the agents are not shown here for space reasons.

For the AGVTS, the TransportAgent will play the role of initiator, the AGVAgents will be the participants. When studying the structures for both concerns, it becomes clear that



```
<archComposition id="CNETExtended">
  <description>CNET extended with internals
      CNETConnector</description>
  <substructure id="CNETConnectorInternalStructure">
    <subject>
      <archStructure href="#CNET">
      <element href="#CNETConnector">
    </subject>
    <archStructure href="#CNETConnectorInternal">
    <interfaceMapping>
      <outerInterface href="#CNETConnector.initiator">
      <innerInterface href="#InitiatorStub.initiator">
    </interfaceMapping>
    <interfaceMapping>
      <outerInterface href="#CNETConnector.participant">
      <innerInterface href="#ParticipantStub.participant">
    </interfaceMapping>
  </substructure>
</archComposition>
```

Fig. 8. (a) CNET structure. (b) CNETConnectorInternal structure. (c) Composition specification to compose CNET and CNETConnectorInternal. (d) Reification of composition, called CNETExtended.

composing the structures is not straightforward since there is a mismatch. The `Initiator` and `Participant` components can not be mapped on the `TransportAgent` and `AGVAgent` components respectively because there are no matching interfaces. Additionally, the environment offer support for communication between agents by offering the `Environments. communication` interface, independent of the underlying network facilities or physical distribution. Thus, introducing a `CNETConnector` as it is between `AGVAgent` and `TransportAgent` is not a good idea, the communication between the agents should go through the environment. To solve this mismatch, we define a substructure for the `CNETConnector`.

Step 2: Composition of agents/environment and coordination. Figure 8(b) shows the internal structure of the CNETConnector. We introduce two stubs to take care of translating the communication acts in Contract Net to messages that can be send through a communication subsystem. More concretely, the components will do message assembly/deassembly, message encoding/decoding, etc. The communication sub system is an abstract representation of a kind of network infrastructure.

We have used the substructure composition operator in fig. 8(c) to define that the CNETConnectorInternal structure splits the `CNETConnector` of the CNET structure. The first part of the substructure operator describes a subject, either a component or a connector, by providing a reference to the associated structure and element. Next, the operator contains a reference to the structure that describes the substructure for the subject. Finally, there are one or several interfaces mappings, describing the relation between an interface of the subject (outer interface) and the substructure (inner interface). Figure 8(d) shows the reification of this composition. The large box around `ParticipantStub` and several other components represents the `CNETConnector`. The two interfaces on the border of this box are both interfaces of the `CNETConnector` and the `ParticipantStub` and `InitiatorStub` respectively.

```
<archComposition id="AgentsAndCoordination">
 <description>Composition between Agents and CNET
concerns</description>
 <mapping>
  <subjects>
   <archStructure href="#CNETExtended">
   <elements selectionStatement="Participant*">
  </subjects>
  <target>
   <archStructure href="#AgentEnvironmentStructure">
   <element href="#AGVAgent">
  </target>
  <interfaceMapping>
   <outerInterface href="#AGVAgent.communication">
   <innerInterface href="#ParticipantStub.communication">
  </interfaceMapping>
 </mapping>
 <mapping>
  <subject>
   <archStructure href="#CNETExtended">
   <elements selectionStatement="Initiator*">
  </subject>
  <target>
   <archStructure href="#AgentEnvironmentStructure">
   <element href="#TransportAgent">
  </target>
  <interfaceMapping>
   <outerInterface href="#TranportAgent.communication">
   <innerInterface href="#InitiatorStub.communication">
  </interfaceMapping>
 </mapping>
 <unification>
  <subject>
    <archStructure href="#CNETExtended">
    <element href="#ComSubSys">
  </subject>
  <target>
    <archStructure href="#AgentEnvironmentStructure">
    <element href="#Environment">
  </target>
 </unification>
</archComposition>
```

Fig. 9. Composition specification to compose CNETExtended and AgentEnvironment.

Finally, we can compose the resulting CNETExtended structure with the AgentEnvironment structure in fig. 9. The composition specification uses three composition operators: two times mapping and one time unification. The mapping composition operator specifies subjects, a target and specifies one or several mappings between interfaces of the target (outer interface) and subjects (inner interface). In this case, `Participant` and `ParticipantStub` are mapped on the `AGVAgent`, `Initiator` and `InitiatorStub` on the `TransportAgent`, the join points are the composition interfaces. Notice the special selection statement in the specification. The star stands for one or several characters and this statement selects multiple elements based on their name. Unification specifies a subject and a target. We make a distinction between subject and target because we want to make clear which of them will be part of the resulting structure. In this case, we unify the `ComSubSys` with the `Environment`.

Step C3: Centralized subsystems. The goal of the first distribution schema is to allocate all agents and the environment to a single server. One of the problem to solve with such single server is that `Ensor` is situated on the AGV vehicle, thus remote access is needed to this component. To solve this problem, we define two subsystems in fig. 10, `Controller` and `RemoteEnsor`, to be deployed on single server and AGV vehicle respectively. Both

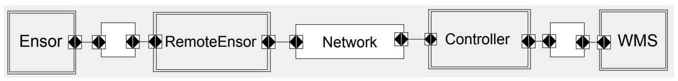
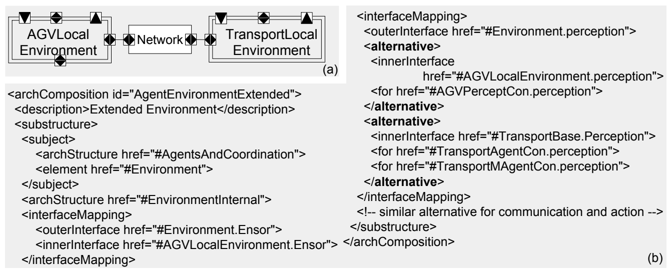
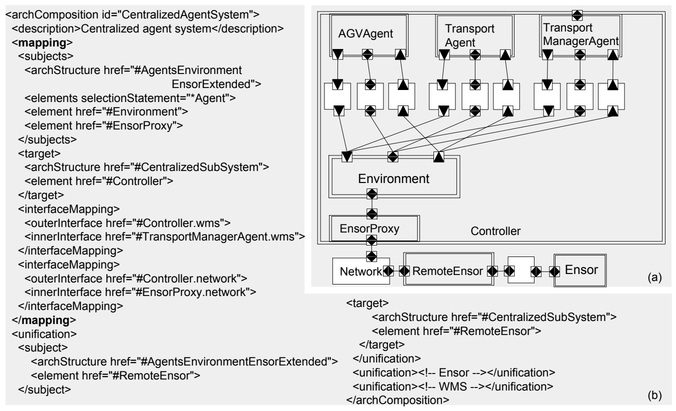


Fig. 10. CentralizedSubSystem structure



```
<archComposition id="AgentEnvironmentExtended">
  <description>Extended Environment</description>
  <substructure>
    <subject>
        <archStructure href="#AgentsAndCoordination">
        <element href="#Environment">
    </subject>
    <archStructure href="#EnvironmentInternal">
    <interfaceMapping>
        <outerInterface href="#Environment.Ensor">
        <innerInterface href="#AGVLocalEnvironment.Ensor">
    </interfaceMapping>
    <interfaceMapping>
      <outerInterface href="#Environment.perception">
      <alternative>
        <innerInterface
                  href="#AGVLocalEnvironment.perception">
        <for href="#AGVPerceptCon.perception">
      </alternative>
      <alternative>
        <innerInterface href="#TransportBase.Perception">
        <for href="#TransportAgentCon.perception">
        <for href="#TransportMAgentCon.perception">
      </alternative>
    </interfaceMapping>
    <!-- similar alternative for communication and action -->
  </substructure>
</archComposition>
```

Fig. 11. (a) EnsorConnectorInternal structure. (b) Composition specification.



```
<archComposition id="CentralizedAgentSystem">
 <description>Centralized agent system</description>
 <mapping>
  <subjects>
   <archStructure href="#AgentsEnvironment
                                   EnsorExtended">
   <elements selectionStatement="*Agent">
   <element href="#Environment">
   <element href="#EnsorProxy">
  </subjects>
  <target>
   <archStructure href="#CentralizedSubSystem">
   <element href="#Controller">
  </target>
  <interfaceMapping>
   <outerInterface href="#Controller.wms">
   <innerInterface href="#TransportManagerAgent.wms">
  </interfaceMapping>
  <interfaceMapping>
   <outerInterface href="#Controller.network">
   <innerInterface href="#EnsorProxy.network">
  </interfaceMapping>
 </mapping>
 <unification>
  <subject>
    <archStructure href="#AgentsEnvironmentEnsorExtended">
    <element href="#RemoteEnsor">
  </subject>
  <target>
    <archStructure href="#CentralizedSubSystem">
    <element href="#RemoteEnsor">
  </target>
 </unification>
 <unification><!-- Ensor --></unification>
 <unification><!-- WMS --></unification>
</archComposition>
```

Fig. 12. Final composition in centralized agent system for AGVTS. (a) Composition specification. (b) Reification of composition.

have an interface for remote communication (connected to each other). The `Controller` is connected to the `WMS`, `RemoteEnsor` to the `Ensor` system.

Next, we want to define the relation between CentralizedSubsystems and the remainder of the application. There is a mismatch, since the `Environment` component is not suited for remote communication with `Ensor`. To solve this problem, we define a substructure for the `EnsorConnector`.

Step C4: EnsorConnectorInternal and composition. Figure 14(a) shows the internal structure of the `EnsorConnector`, containing two subcomponents connected with a connector, respectively `EnsorProxy` and `RemoteEnsor`. `EnsorProxy` will act as a proxy for `Ensor` from the perspective of the `Environment` component. Again, we use the substructure operator to relate the EnsorConnectorInternal structure with the AgentEnvironment structure in fig. 14(b).

Finally, we can compose the CentralizedSubSystem structure with the ExtendedAgentEnvironment structure in fig. 12. The `EnsorProxy` becomes a subcomponent of the `Controller`, together with all agents and the environment component. We unify the pairs of `RemoteEnsor`, `Ensor` and `WMS` components with each other

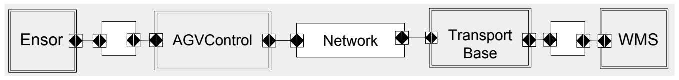
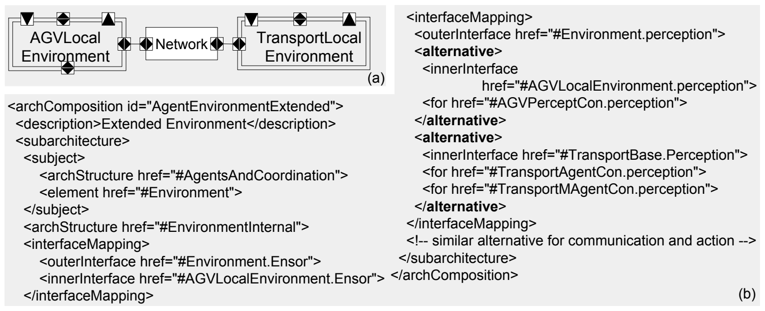


Fig. 13. DecentralizedSubSystem structure.



```
<archComposition id="AgentEnvironmentExtended">
  <description>Extended Environment</description>
  <subarchitecture>
    <subject>
      <archStructure href="#AgentsAndCoordination">
      <element href="#Environment">
    </subject>
    <archStructure href="#EnvironmentInternal">
    <interfaceMapping>
      <outerInterface href="#Environment.Ensor">
      <innerInterface href="#AGVLocalEnvironment.Ensor">
    </interfaceMapping>
    <interfaceMapping>
      <outerInterface href="#Environment.perception">
      <alternative>
        <innerInterface
                href="#AGVLocalEnvironment.perception">
        <for href="#AGVPerceptCon.perception">
      </alternative>
      <alternative>
        <innerInterface href="#TransportBase.Perception">
        <for href="#TransportAgentCon.perception">
        <for href="#TransportMAgentCon.perception">
      </alternative>
    </interfaceMapping>
    <!-- similar alternative for communication and action -->
  </subarchitecture>
</archComposition>
```

Fig. 14. (a) EnvironmentInternal structure. (b) Specification to compose EnvironmentInternal with AgentsAndCoordination.

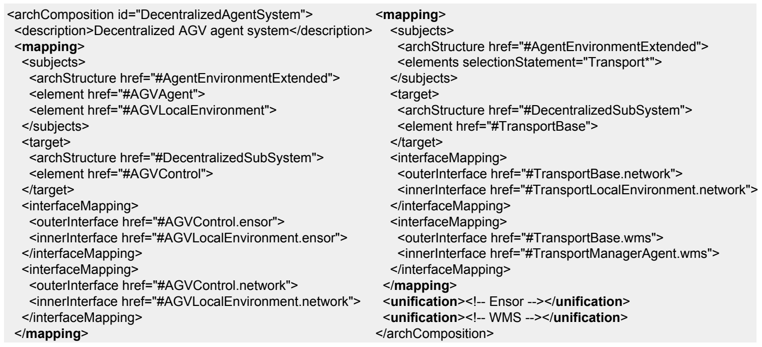
Step D3: Decentralized subsystems. The goal of the second distribution schema is to distribute the agents. In such a schema, the `AGVAgent` is allocated on the AGV vehicle, the `TransportAgent` and `TransportManagerAgent` on one or several servers. To allow this we defined two subsystems in fig. 13, `AGVControl` and `TranportBase`, to deploy on the AGV vehicle and a server respectively. Both have an interface for remote communication (connected to each other). The `TranportBase` has a connection to the `WMS`, the `AGVControl` component to `Ensor`.

Next, we want to define the relation between DecentralizedSubsystem structure and the remainder of the application. Again, there is a mismatch between structures. Because the agents should be located on different subsystem (allocated on different computers), the environment component must be split. We define a substructure in the next paragraph.

Step D4: EnvironmentInternal and composition. Figure 14 shows the internal structure of the environment. The environment is split up in two local environment, `AGVLocalEnvironment` and `TransportLocalEnvironment`. These two local environments form a virtual environment, by synchronizing state and taking care of communication of the network without involving the agents.

Defining the composition with this substructure is less trivial then the previous examples. The Environment is split in two components, but three interfaces (perception, communication, action) are offered by both components, making a simple interface mapping impossible. To solve this problem, we provide several alternative mappings and specify for which connector interface the alternative is. Figure 14(b) shows the composition specification.

Finally, we can compose the system together to form a decentralized version of the AGVTS. This system will look exactly like the system described in fig. 5. Figure 15 contains the composition description which is similar to previous composition specifications.



```
<archComposition id="DecentralizedAgentSystem">
 <description>Decentralized AGV agent system</description>
 <mapping>
  <subjects>
   <archStructure href="#AgentEnvironmentExtended">
   <element href="#AGVAgent">
   <element href="#AGVLocalEnvironment">
  </subjects>
  <target>
   <archStructure href="#DecentralizedSubSystem">
   <element href="#AGVControl">
  </target>
  <interfaceMapping>
   <outerInterface href="#AGVControl.ensor">
   <innerInterface href="#AGVLocalEnvironment.ensor">
  </interfaceMapping>
  <interfaceMapping>
   <outerInterface href="#AGVControl.network">
   <innerInterface href="#AGVLocalEnvironment.network">
  </interfaceMapping>
 </mapping>
<mapping>
  <subjects>
   <archStructure href="#AgentEnvironmentExtended">
   <elements selectionStatement="Transport*">
  </subjects>
  <target>
   <archStructure href="#DecentralizedSubSystem">
   <element href="#TransportBase">
  </target>
  <interfaceMapping>
   <outerInterface href="#TransportBase.network">
   <innerInterface href="#TransportLocalEnvironment.network">
  </interfaceMapping>
  <interfaceMapping>
   <outerInterface href="#TransportBase.wms">
   <innerInterface href="#TransportManagerAgent.wms">
  </interfaceMapping>
 </mapping>
 <unification><!-- Ensor --></unification>
 <unification><!-- WMS --></unification>
</archComposition>
```

Fig. 15. Final composition in decentralized agent system for AGVTS. (a) Composition specification. (b) Reification of composition.

4.3 Discussion

Structural Crosscutting Relations made Explicit. Using architectural structures to separate concerns and add explicit compositions of structures proved to be valuable from different perspectives. By using separate structures, an architect can effectively separate several important concerns. With the composition operators, an architect can explicitly define the relations between different concerns. As discussed in the previous section, the combination of separation and explicit relations helps to improve the understandability and changeability of individual architecture concerns, such as in the distribution-specific scenario in the AGVTS architecture. It also became easier to compare two different deployment schemas for the same decomposition into agents/environment and coordination. Notice that the final composition of the AGVTS, the decentralized version, looks the same as the original crosscutting design. The main difference is the process the architecture needs to undertake to get into this structure. The new architecture decomposition has clearly separated the crosscutting structural concerns and provided explicit composition specifications between these structures.

Tool Support. Making the structural composition specification part of the architectural description is one of the main innovations of this paper. Because the composition is clearly specified in XML, appropriate tool support can make reification really easy. Further research is required on this point. A drawback of the new design is that it may be more difficult to have an overall view on the system. While the structures of individual architectural concerns is much clearer, some reification of compositions must be included in the architectural documentation to get a good overall view of the system. Again, tool support would be essential to address this issue.

Relation with other Development Stages. Obviously, we have investigated the advantages of separating concerns considered during architectural design. As such, choosing the appropriate architectural concerns is very important and largely depend on the application context and its stakeholders. Also, we did not explicitly study the relations with previous and following development phases. Investigating the exact relations with concerns identified in requirements and concerns tackled during detailed design remains to be done.

Composition Operators and other ADLs. Since the idea behind the composition operators only relies on basic ADL elements, such as components, interfaces and connectors, integration with for example the ACME ADL should be straightforward. Of course, the syntax of the operators depends on the ADL metamodel itself and needs to be adapted. However, integration with ACME ADL is a matter of our future work. The difference between existing architectural relations (connectors, links or even aspectual binding [22]) and the composition operators presented here is that the former describe relations between *components or connectors* while the latter describe relations between *several structures*.

Operators Applicability. The composition operators rely on specifying corresponding interfaces (interface mapping). This restriction to one-to-one mapping might not work well in some cases. For example, one structure could contain two interfaces who map on one interface in another structure. Another example is the problem with the substructure in step D4, in which an interface occurs on two subcomponents. The latter problem is solved by adding an alternative specification to substructure, but this might not always be possible. In some situations there will be no other choice but to refactor one the interfaces of one of the structures. Our composition operators are not the only possible operators. We did choose them to solve overlap and mismatches between architectural structures. Together, they are a sufficient set to compose the separate structures in the AGVTS architecture. Other possible operators could include operators to solve interferences between names, operators to solve non-corresponding interfaces, etc. Finally, quantification [19] over architectural elements is found in the section statement for the elements specification. In the example, the expressions are rather limited (text and wildcards), but they can easily be extended. Finally, we only considered a representative part of the design of the AGVTS. How scalable the approach is in case of a large increase in concerns and structures needs further exploratory studies. Indeed, this paper provides a first significant contribution in this respect.

5 Related work

The outcomes of some recent empirical studies have pointed out that crosscutting-related problems relative to certain architectural concerns, such as exception handling [18], persistence, and distribution [29], can manifest early in design decompositions.
Some AO techniques for programming (e.g. HyperJ [40]) and requirements engineering (e.g. [10, 33, 39]) provide structural composition operators for enabling the separation of certain recurring crosscutting concerns. Surprisingly, composition mechanisms for modularizing architectural crosscutting structures have been neglected in most of the available literature [5, 13] focusing on discussing the interplay of aspects and ADLs. Batista and colleagues [5] have identified seven issues related to the appropriateness of abstractions in conventional and AO ADLs for representing crosscutting concerns. However, the analyzed composition-related issues mostly focus on to what extent connectors and attachments allow for behavior-dependent aspect composition. In addition, existing AO ADLs [22, 25, 34, 37, 35, 36] expose join points as nodes in a dynamic component or object call graph. As illustrated in this paper, this begs the question as to how structural crosscutting concerns can be modularized by behavioral composition mechanisms for software architectures.

In the field of software architecture, there is little support for relations between structures or views. Rozanski and Woods [38] identifies that quality concerns crosscutting several architectural structures. The authors introduce architectural perspectives as complementary to structures in the sense that they define a set of activities, tactics and guidelines to ensure that the structures fulfil a quality. IEEE-Std-1471 [26] provides an strong conceptual model relating stakeholders, concerns and views, but lacks a way to describe relations between different views. Clements et al. [12] provides some support for relations between views by integrating information beyond views in the form tables and an associated informal description.

6 Conclusion

The lack of explicit relations between structures in an architectural description is a severe problem, leading to improper documentation of concerns that crosscut different structures. This paper illustrated that several concerns crosscut the architecture structures of an industrial automatic transportation system. Based on the analysis of the software architecture, we identified overlap and structural mismatches as main challenges to compose structures. We proposed an initial set of structural composition operators in xADL to cope with these challenges, namely substructure, mapping and unification.

To illustrate the feasibility, we have used the composition operators to refactor the architecture of the transportation system and showed that the composition operators allowed to easily change the AGVTS architecture with respect to distribution. Within the limited setting of this paper, the results look promising. As future work, we plan to formally extend xADL by integrating the composition operators in the language definition and to investigate tool support to apply the compositions.

Acknowledgments. Nelis is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Nelis and Alessandro are also partially supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on Aspect- Oriented Software Development (AOSD-Europe), 2004-2008.

References

1. Xslt. `http://www.w3.org/TR/xslt`.
2. J. Araujo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan. Version 1 of the early aspects landscape report. Published on www.early-aspects.net, 2005.
3. E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Software*, January/February, 2006.
4. L. Bass, P. Clements, and R. Kazman. *Software Architectures in Practice (Second Edition)*. Addison-Wesley, 2003.
5. T. Batista, C. Chavez, A. Garcia, C. Sant'Anna, U. Kulesza, A. Rashid, and F. Filho. Reflections on architectural connection: Seven issues on aspects and adls. In *Workshop on Early Aspects held at ICSE'06*, 2006.
6. N. Boucké and T. Holvoet. Relating architectural views with architectural concerns. In *Early Aspect workshop at ICSE*, 2006.
7. N. Boucké, T. Holvoet, T. Lefever, R. Sempels, K. Schelfthout, D. Weyns, and J. Wielemans. Applying the Architecture Tradeoff Analysis Method (ATAM) to an industrial multi-agent system application. Technical Report CW431, Dept. of Computer Sience, KULeuven, 2005.
8. N. Boucké, D. Weyns, and T. Holvoet. Experiences with Theme/UML for architectural design of a multiagent system. In *Multiagent Systems and Software Architecture, Proceedings of the Special Track at Net.ObjectDays 2006*, pages 87–110. Net.ObjectDays, K.U.Leuven, 2006.
9. N. Boucké, D. Weyns, K. Schelfthout, and T. Holvoet. Applying the ATAM to an architecture for decentralized control of a transportation system. In *Quality of Software Architectures conference (QoSA)*, volume LNCS 4214, 2006.
10. R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *Aspect Oriented Software Development conf.*, 2007.
11. S. Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.
12. P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. Addison Wesley, 2003.

13. C. Cuesta, M. Romay, P. Fuente, and M. Barrio-Solorzano. Architectural aspects of architectural aspects. In *European Workshop on Software Architecture (EWSA)*, volume LNCS 3527, pages 247–262, 2005.
14. E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, .14(2):199–245, 2005.
15. Eclipse. `www.eclipse.org`.
16. Aspectj project. `www.eclipse.org/aspectj/`.
17. Egemin. Egemin website. `www.egemin.com`.
18. F. Filho, N. Cacho, R. Ferreira, E. Figueiredo, A. Garcia and C. Rubira. Exceptions and Aspects: The Devil is in the Details. In *Proc. Int. Conf. on Foundations on Software Engineering*, 2006.
19. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA*, 2000.
20. R. E. Filman, T. Elrad, S. Clarke, and M. Akṣit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.
21. FIPA-Standard. Fipa Contract Net interaction protocol specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`, December 2002.
22. A. Garcia, C. Chavez, T. Batista, C. Sant'anna, U. Kulesza, A. Rashid, and C. Lucena. On the modular representation of architectural aspects. In *Proc. of the European Workshop on Software Architecture*, 2006.
23. A. Garcia and C. Lucena. Taming heterogeneous agent architectures with aspects. In *Communications of the ACM (Accepted for publication)*, 2005.
24. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
25. J. Grundy. Multi-perspective specification, design and implementation of components using aspects. *Int. Journal of Software Engineering and Knowledge Engineering*, 10(6), 2000.
26. IEEE. Recommended practice for architectural description of software-intensive systems (ansi/ieee-std-1471), September 2000.
27. ISR, Institute for Software Reseach. Archstudio 4.0 tool set for the xadl language. `http://www.isr.uci.edu/projects/archstudio/`.
28. S. Kojarski and D. H. Lorenz. Modeling aspect mechanisms: A top-down approach. In *In Proc. Int. Conf. on Software Engineering*, pages 212–221, 2006.
29. U. Kulesza, C. SantAnna, A. Garcia, R. Coelho, A. von Staa and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. of Int. Conf. on Software Maintenance*, 2006.
30. N. Loughran, A. Rashid, R. Chitchyan, N. Leidenfrost, J. Fabry, N. Cacho, A. Garcia, F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Boucké, T. Holvoet, A. Jackson, A. Nedos, N. Hatton, J. Munnelly, S. Fritsch, S. Clarke, M. Amor, L. Fuentes, M. Pinto, and C. Canal. A domain analysis of key concerns known and new candidates. AOSD-Europe Deliverable D43, AOSD-Europe-KUL-6, 2006.
31. N. Medvidovic. Moving architectural description from under the technology lamppost. In *Proc. EUROMICRO Conf. on Software Engineering and Advanced Applications*, pages 2–3. IEEE Computer Society, 2006.
32. N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
33. A. Moreira, A. Rashid, and J. Araujo. Multi-dimensional separation of concerns in requirements engineering. In *Int. Conf. on Requirements Engineering (RE'05)*, pages 285–296, 2005.
34. A. Navasa, M. Pérez, J. Murillo, and J. Hernández. Aspect oriented software architecture: a structural perspective. In *Workshop on Early Aspects, AOSD2002*, 2002.
35. J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro. Prisma: Towards quality, aspect oriented and dynamic software architectures. In *Int. Conf. On Quality Software*, 2003.
36. N. Pessemier, L. Seinturier, and L. Duchien. Components, adl and aop: Towards a common approach. In *Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution*, 2004.
37. M. Pinto, L. Fuentes, and J. M. Troya. A dynamic component and aspect-oriented platform. *Computing Journal*, 48(4):401–420, 2005.
38. N. Rozanski and E. Woods. *Software Systems Architecture*. Addison Wesley, 2005.
39. L. Silva. An aspect-oriented approach to model requirements. In *Proc. of the Doctoral Consortium on Requirements Engineering (with RE conference)*, 2005.
40. P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Int. Conf. on Software Engineering*, pages 107–119, 1999.
41. D. Weyns and T. Holvoet. A reference architecture for situated multiagent systems. In *Postproc. of Int. Workshop on Environments for Multiagent Systems*, volume LNCS 4389, 2006.
42. D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for multiagent systems: State-of-the-art and research challenges. In *Revised papers of the E4MAS workshop at AAMAS'04*, volume LNCS 3374, 2005.