Composing Structural Views in xADL

Nelis Boucké¹, Alessandro Garcia², and Tom Holvoet¹

¹ DISTRINET, KULEUVEN

{nelis.boucke,tom.holvoet}@cs.kuleuven.be

² Computing Department, Lancaster University

garciaa@comp.lancs.ac.uk

Abstract. Experience with building an architecture for an industrial Automatic Guided Vehicle Transportation System (AGVTS) shows that several essential concerns crosscut the architectural views. To cope with this, a stronger separation proved to be necessary, i.e. using different views for different concerns. In practice this was difficult, since the support for relations between views is very limited. This makes separation of concerns in views hard, thereby increasing maintenance overhead and reducing reuse capabilities. Our claim is that specifying compositions of views is as important as specifying the views itself. This paper extends a representative architectural description language (xADL) which support for composing structural views, by introducing three relations, namely *refinement*, *mapping*, and *unification*. Improving separation of concerns in views and their explicit composition enhances architecture understandability and changeability. The feasibility of the relations is assessed by redesigning the AGVTS architecture. Based on a real maintenance scenario, we investigate to what extent these explicit compositions lead (or not) to enhanced architectural changeability for evolving the distribution strategy in the AGVTS system.

1 Introduction

The architecture of a software system defines the design structure or structures, which comprise software elements, the externally visible properties of those elements, and the relationships among them [3]. This implies that the core issue in architectural design is to define and compose the high-level design structures that are relevant to key stakeholders' concerns. Over the last decades several Architectural Description Languages (ADLs) have been proposed to describe architectures.

Experience with building a multiagent architecture (MAA) for an industrial Automatic Guided Vehicle Transportation System (AGVTS) shows that several concerns typically crosscut the architectural views. A crosscutting concern in an ADL description is a concern that is not effectively modularized using the abstractions of that ADL, in this case the modularization at the level of architectural views. Using MAA means structuring the system in several autonomous entities (agent components), connected to an environment component, working together (coordinating) to achieve the goals of the system [46]. We have observed that some architecturally-relevant concerns, such as control, coordination, and distribution, crosscut the decomposition in architectural views of the AGVTS system. The architects judged that a stronger separation was necessary to allow evolving individual concerns more easily and targeted a clean separation of the description of different concerns in different views.

In order to achieve the separation of such crosscutting architectural concerns, we encountered a fundamental problem that is the subject of this paper. Having divided to conquer, we must reunite to rule [29]. Once the concerns are separated using the views, there is a need to integrate the views to obtain the final system, and this is where the current architectural practice falls short. For integration, we need to define *relations* between views. In this context, we use the term *composition* for an aggregation of several views and relations. As a consequence, the *system* is a composition of all views and associated relations between the views.

Looking to architectural practice, there is good support to describe several views, but the support for describing relations between views is limited. For example, Clements and colleagues [12] use tables to define a mapping between elements in different views. Yet, the relation remains informal (relying on a textual explanation) and the support for different types of relations is too limited. The IEEE-1471 standard[27] requires that any inconsistencies between views are documented, but leaves completely open how this should happen. In other approaches the relations between views remain implicit or informal. Without proper support for relations, the architect has no choice but to leave the concerns mingled with each other in the views. Not separating important concerns makes it harder to comprehend the architecture and leads to increased maintenance and change overhead, reduced reuse capabilities, and generally results in architectural erosion over the system lifetime [4].

Our claim is that integrating the views is as important as specifying the views itself. Relations and compositions must become first class, and become an explicit and integral part of an ADL. Based on these observations, we extend a representative ADL with explicit support for composing structural views. In this first extension we limit ourselves to add relations between one type of view only, namely the structural view of xADL [14]. xADL is a general-purpose, extensible ADL to model architecturally-relevant structures. xADL supports the specification of structural views, but there is little to no support for the relations between views. Concretely, we introduce three types of relations, *refinement*, *mapping*, and *unification*, and use these relations to compose several structural views. The xADL language is used as an example, but similar composition rules could be of use to other ADLs as well. To assess the feasibility of the operators, we use them to refactor and evolve the AGVTS architecture in order to make the key concerns of the architect explicit, together with the associated structural views and relations between the views.

The proposed approach differs from existing aspect-oriented ADLs by focussing on composition of structural views. The current state of art in aspect-oriented ADLs concentrates on defining aspectual components and aspectual relations to add or change *behavior* on component interfaces, starting from the decomposition into components and connectors [4,37,39]. In another words, they tend to mimic AO composition mechanisms supported by the family of AspectJ [16]-like programming languages [32]. This paper focuses on the structural impact of a concern, starting from a decomposition into structural views and trying to compose these views together.

Overview: The remainder of this paper is structured as follows. Section 2 introduces ADLs. Section 3 introduces our case study and illustrates the lack of relations in a more concrete context. In section 4 we introduce compositions and relations in xADL and

review an excerpt of the AGVTS in the context of these relations. Section 5 describes related work. Finally, we conclude in section 6.

2 Architectural Description Languages

In the last decades, several ADLs have been proposed to model architectures. ADLs exist in all kinds of shapes and forms. Several ADLs have a formal foundation (like ACME [24], xADL [15] and AADL [42]), but we also consider less formal approaches to define an ADL, like [12] or [40]. Some ADLs are meant to model a particular application domain, others are more general-purpose. We will use the xADL language in our illustrations. The motivation to use the xADL language is that it provides a core language with a simple set of general-purpose constructs to model an architecture. Additionally, it provides an easy way to extend the language definition and the associated tool support for new concepts, allowing fast prototyping of new ADL constructs.

xADL supports three types of views: (1) a view defining component types, called types or `archType` in xADL; (2) structural views describing component and connector types and how they are linked to each other, called architectural structure or `archStructure` in xADL; (3) instance views describing component and connector instances and how they are linked to each other, called architectural instances or `archInstance` in xADL. Both second and third are a kind of component and connector viewtype in the sense of [12]. The second one shows configuration types of components and connectors emphasizing on how they should be linked together (called structure by the xADL authors), while the third one shows component and connector instances to document the dynamics of the system. In this paper, we focus on the second type of view of xADL and call this a structural view on the system, inline with the terminology suggested by the xADL authors.

2.1 Basic Elements in xADL

An structural view in xADL is built around the following basic elements¹.

- **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 a unique identifier, a textual description, and a set of interfaces.
- **Interfaces:** Interfaces are components' and connectors' portals to the outside world. A "interface" in xADL is called "port" for components and "roles" for connectors in other languages like ACME ADL. Interfaces have a unique identifier, a textual description, and a direction. The direction indicates whether the interface is provided, required, or both.
- **Links:** Links are connections between interfaces, defining the topology of the architecture.

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 [15] to improve readability of the specification. We will only once include a key, the other figures use the same graphical notation.

Figure 1a shows an example architectural description containing two components (Agent and Environment), one connector ActionCon, and links to attach the connector to the components. The environment in this picture is a software component providing operations to the agents to view and manipulate the world. In principle, the identifier of an interface can be any unique string. We follow the convention that the identifier is built following the pattern 'element-name-dot-interface-name', like Agent.action. References are represented by the href property, containing a unique identifier or an element after the sharp symbol.

xADL is supported by a tool set integrated in the Eclipse platform, called ArchStudio. We used ArchStudio 4.0, available since November 2006 [28]. ArchStudio contains both a visual editor, that allows to graphically manipulate the architecture (notation similar to fig. 1b) and a special-purpose specification editor, representing the architecture in the form of a tree.

Special Construct: Subarchitecture. xADL has an additional construct called *subarchitecture*, which we would like to explain here since one of our relations is inspired by it. 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. We briefly describe types in xADL because they are used

¹ Based on [15]. The group element is left out because it is not used in this paper.

```

<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">
          <!-- As in previous figure -->
        </archStructure>
        <SignatureInterfaceMapping id="ControlS">
          <outerSignature href="#SystemType.control">
            <innerInterface href="#Environment.control">
              </innerInterface>
            </outerSignature>
          </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">
            </signature>
          </interface>
        <type href="#SystemType">
          </type>
        </component>
      </archStructure>
    </xArch>

```

Fig. 2. Example with subarchitecture construct

the subarchitecture construct, but we will not come back to types in the remainder of this paper. 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. A signature prescribes what interface a component must offer to be of the type. Types are typically used if exactly the same component or connector type is needed in several structural views.

Figure 2 contains an example component type `SystemType`, having one signature called `SystemType.control`. The `SystemStructure` architectural structure uses the `SystemType` to construct the `System` component, and provides a href 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 an (indirect) relation between two architectural structures: `SystemStructure` is related with `AgentEnvironment` through its `System` component of type `SystemType`.

3 Case Study: Automatic Guided Vehicle Transportation System

This section introduces the industrial case study, an Automatic Guided Vehicle Transportation System (AGVTS). First, the application is briefly introduced in section 3.1. Next, we exploit an excerpt from an existing architectural description [5] to illustrate the problem of concerns that crosscut the structural views in section 3.2.

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 uses an external software system – called Ensor – 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 the AGVTS 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 coming from a Warehouse Management System (WMS).

Multiagent Architecture. In a joint research project between DistriNet and Egemin [17], a producer of automated logistic systems, the AGVTS was built with a MAA. This architecture decentralizes control and answers to new market demand for flexible, open and robust systems that can handle dynamic operating conditions in an autonomous way. These qualities are ascribed to MAAs [7].

Using a MAA means structuring the system in several autonomous entities (agent components) connected to an environment component. The agents and environment are built according to a reference architecture for multi-agent systems [44]. An agent component operates autonomously, but coordinates with other agent components 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 [45]. The environment provides interfaces for perception, action and communication to the agents.

Concerns. In this paper, we will focus on three essential concerns for the software architect building an MAA for an AGVTS, namely control, coordination and distribution. The concerns are identified based on the experience of the architects building an AGVTS and each of them has a significant impact on the structural view of the system. We briefly discuss each of the concerns and describe the architectural implications of the respective concerns before going into details on crosscutting concerns in the next section.

The control concern covers the decomposition into agents and an environment (to realize decentralized control in the AGVTS). We consider this decomposition as the dominant decomposition in an MAA. In the architectural description we will mainly focus on the interaction between the agents and the environment and less on the internals of the respective components. In the AGVTS we identified three types of agents. There is an AGVAgent for each AGV vehicle, responsible for managing the tasks of an AGV. There is a Transport-Agent for each transportation task in the system, responsible for searching an appropriate AGV to perform a new transport and to follow up the dispatching of the tasks. Finally, there are one or more TransportManagerAgents, in charge of communicating with a WMS and managing (create, remove, etc.) the TransportAgents. For an AGVTS, the tasks of the Environment include providing the interfaces for perception, action and communication to the agents and the translation of high-level commands from the AGVAgent to Ensor, the software package for low-level control of the AGV vehicle.

Because agents operate autonomously, coordinating their behavior is essential to let the multiagent system operate as a whole. Because coordination is so important in the AGVTS, the architects decided to separate this concerns. The coordination concern

Fig. 3. First decomposition view of an existing architectural description of the AGVTS, to illustrate crosscutting of the control and distribution concern. Same graphical notation as fig. 1.

covers all types of coordination between different agents in the system, e.g. assigning transport or avoiding collisions on crossroads.

The third concern, distribution, covers the implications of building a software system on a physically distributed infrastructure. We focus on the decomposition into subsystems to allow easy deployment on a physical infrastructure and any additional support needed for this, like remoting or synchronization support. Note that we specifically look for the impact of distribution on the *structure* of the software (thus in structural views), independent of the deployment view. Distribution is specifically important for an AGVTS, since the physical infrastructure exists of an industrial computer on each AGV vehicle and one or several server systems, connected through a wireless network.

3.2 Concerns Crosscutting Structural Views

As defined in the introduction, a crosscutting concern in an ADL description is a concern that is not effectively modularized using the abstractions of that ADL [4]. Any type of decomposition implies that there are concerns that crosscut with that decomposition. If an architect decides to decompose the system in several structural views, there will be concerns that crosscut these structural views.

To illustrate crosscutting and the need for a stronger separation we start from the original architectural documentation and show how the concerns control, distribution and coordination crosscut in the architectural views of the AGVTS.

Control and Distribution. Figure 3 shows the first decomposition view of the architectural description of the AGVTS. Careful analysis reveals that this decomposition mixes up at least two of the concerns: control and distribution.

To understand better which part of the decomposition is done for which concerns, fig. 4 shows separate structural views per concerns. The left part shows the Control view, outlining a decentralized control architecture defining

Fig. 4. Left: Control view, showing the decomposition into agents and environment. Right: Distribution view, showing the decomposition into subsystems for deployment. Same graphical notation as fig. 1.

the types of agents and an environment. The right part shows the Distribution view, defining two subsystems (AGVControl and TransportBase), to be deployed on an AGV vehicle and a server respectively. Note that the Environment component in the Control view corresponds to the AGVLocalEnvironment and TransportLocalEnvironment components in the StructuralView. But in the StructuralView it is implicit which part belongs to which concern and what the relations between different concerns are, mingling up the two concerns. Additionally, there are several other views mingling up the same concerns, leading to crosscutting.

In the introduction some well-known drawbacks of crosscutting concerns are mentioned. Here we illustrate one specific consequence, namely that it is harder to change the architecture with respect to one of the concerns. The architects encountered this problem during a revision the AGVTS architecture. The original architecture located an agent on each AGV vehicle. During the project Egemin realized that switching to a distributed MAA is a big step with far-reaching effects for the company, not only for the software but for the whole organization². Therefore, Egemin proposed a stepwise integration starting with an architecture having all agents located on a single computer system with remote access to the AGV vehicles. In principle, the impact of such a revision would be limited since only the distribution concern is changed. Unfortunately, due to the concerns crosscutting the views, they had to adapt most of software architecture and change nearly every individual view.

Control and Coordination. To illustrate that crosscutting concerns have a even more broadly scoped impact on the views, we annotated the table of content of the existing architectural description of the AGVTS with two concerns: control and coordination. Figure 5 shows the annotated table of content, showing both static and runtime views. View names marked in **bold** include fragments of the control concern, underlined names include fragments of the coordination concern. Views that are both **bold** and underlined intermingle the definition of coordination with agents, illustrating that the two

² This is inline with the observations of Bass et al. [3] that an architecture affects the developing organization.

- Static views
 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. 5. List of design models made for the AGV transportation system. Model names in bold include the control concern, underlined model names include the coordination concern.

Fig. 6. Illustration of overlap between both views. The dashed circles point to strange overlap between the elements, for the remainder we use the same graphical notation as fig. 1.

concerns crosscut each other in the views. Notice that this illustration involves several types of view, including static, run-time and deployment types of views. This illustrates the broadly scoped influence of coordination and control on the architectural views.

Relations. Why do the concerns crosscut each other? In the first version of the architecture, the architects did not explicitly consider a set of essential concerns to separate, obviously leading to crosscutting concerns as just illustrated.

Based on the changes to the distribution concerns and the analysis revealing crosscutting concerns, the architects judged that a stronger separation was necessary. During a revision of the architecture, they wanted to separate the description of different concerns in different views. The two views in fig. 4 could be an example of this. That is when the architects encountered the lack of appropriate relations, preventing them from using such strong separation between concerns in views.

The challenges to relate the Control view and Distribution view are:

1. The same element appear in different structural views, e.g. `Ensor`, `WMS` and the respective connectors.
2. Some components must become subcomponents of other components, e.g. `AGVAgent` must become subcomponent of `AGVControl`.
3. If the components would be drawn in the same view, one component would crosscut the border of other components. To illustrate this, we constructed a view mixing the elements of the Control and Distribution view (called `Control&Distribution` view) in fig. 6. One can see that the `Environment` component cuts across the border of the `AGVControl` and `TransportBase` components.

4 Composing Structural Views

This section illustrates how to compose structural views in xADL. Section 4.1 introduces the necessary changes to the xADL language, adding the concepts of composition and introducing three types of relations. Section 4.2 uses the extended language to refactor an excerpt of the AGVTS architecture covering two concerns introduced earlier in this paper: control and distribution.

4.1 Compositions and Relations

Firstly, we add a new element to the xADL description called architectural composition (denoted `<archComposition>` in the XML specification), describing a concrete composition of several structural views and the associated relations. Next, we introduce three types of relations.

- **unification:** Unifies elements (components or connectors) from different structural views 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 structural view on a single element of another structural view (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.
- **refinement:** Defines that a specific structural view (referred to as inner structure) describes a substructure for an architectural component (referred to as outer component) of another structural view. 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.

Figure 7 shows an example architectural composition using the refinement relation. The figure shows three structural views (represented by a UML package symbol: `Struct1`, `Struct2` and `Struct3`) and a composition specification. A composition specification always consists of a unique identifier, a description and a set of relations between the views.

Fig. 7. Example composition using the refinement relation. Struct2 refines component A of Struct1, resulting in Struct3. Same graphical notation as fig. 1.

Fig. 8. Example composition using the mapping and unification relations. Parts of Struct2 are mapped on component A and B of Struct1, resulting in Struct3. Same graphical notation as fig. 1.

In our specific solution, a composition results in a new view that is a reification of the relations and can be used in further compositions (the view has the name of the identifier of the composition). The composition specifies that Struct2 refines component A

Structural view	Purpose	Concern	Step
Control	Describes the agent types, environment and associated relations.	control	1
Centralized	Subsystems needed for centralized deployment.	distribution (centralized)	2C
EnsorConnectorInternal	Describes a refinement of the EnsorConnector.	auxiliary (control&distribution)	3C
ControlExtendedAlt1	Composition of Control and EnsorConnectorInternal	result of composition	3C
CentralizedAGVSystem	Composition of ControlExtendedAlt1 and Centralized	result of composition	4C
Decentralized	Subsystems needed for decentralized deployment.	distribution (decentralized)	2D
EnvironmentInternal	Describes a substructure of the Environment component.	auxiliary (control&distribution)	3D
ControlExtendedAlt2	Composition of Control and EnvironmentInternal	result of composition	3D
DecentralizedAGVSystem	Composition of ControlExtendedAlt2 and Decentralized	result of composition	4D

Fig. 9. The different structural views made for the AGVTS. For simplicity, coordination is not included in the example.

of Struct1. In our approach every results is a new structural view, in the example called Struct3. After the composition, the elements of Struct2 are sub elements (sub components or sub connectors) of component A. The join-points between the structural views are the interfaces, specified in the interface mappings part of the refinement relation. For example, the A.up interface of component A is mapped on interface Z.up of its subcomponent Z.

Currently, the reification of the composition is done by hand. Because the composition specification is specified in XML, appropriate tool support could make applying a composition really easy. Further research is required on this point.

Figure 8 shows an example using the mapping and unification relations, containing three structural views and a composition specification. Mapping and unification are mostly used together in one composition specification, that is why we demonstrate them in this way. The composition specifies that several elements of Struct2 are mapped on (i.e. are subelements of) component A and component B of Struct1, resulting in Struct3. The join-points between the structural views are the interfaces, specified in the interface mapping part of the refinement relation. For example, the B.up interface of component B is mapped on interface X.up of its subcomponent X.

We introduce specifically these three relations to cope with the challenges identified in section 3.2. We briefly recall the challenges and mention the relation coping with the particular challenge.

1. The same element appears in different structural views, e.g. *Ensor*, *WMS* and the respective connectors. This is where *unification* can be used.

2. Some components must become subcomponents of other components, e.g. AGVAgent must become subcomponent of AGVControl. In that case, *refinement* can be used to refine one of the components to prevent this crosscutting.
3. If the components would be drawn in the same view, one component would crosscut the border of other components. This is the type of crosscutting where the relations can cope with. This is where *mapping* can be used.

Notice that there are several important differences between the existing subarchitecture construct (described in section 2.1) and the refinement composition operator. Firstly, subarchitecture defines a relation between a *type* and a structural view. Refinement is a relation between two structural views, resulting in a more direct relation between structural views. Secondly, refinement is not described in the component type, as for subarchitecture, but in a separate composition specification section. This has the advantage that refinement can be varied depending on the context of the target composition.

4.2 Example of Composing Structural Views in the AGVTS

Figure 9 provides an overview of the structural views we have described for the AGVTS, with a short description of the purpose, the associated concern and the step in which the structural view is used. In the concern column we either provide the associated concerns, or mentioned where the view originates from.

The scenario in the AGVTS application is built up as follows. First we define the control concern, by decomposing the system in agents and an environment in step 1. In the next step, distribution is added to the picture, considering two alternative distribution schemas leading to two different systems. Steps 2C to 4C describe a centralized distribution schema, with associated relations to the control concerns. Steps 2D to 4D describe a decentralized distribution schema.

Fig. 10. The Control view showing the agent types, the Environment component and the relations with external components (WMS and Ensor). To simplify the figure, we only showed the names of connectors and interfaces used in the composition specifications. Same graphical notation as fig. 1.

Fig. 11. Centralized view. Same graphical notation as fig. 1.

Fig. 12. Left: EnsorConnectorInternal view. Right: Composition specification to compose the EnsorConnectorInternal (at the left) and the Control view (fig. 10), using the same graphical notation as fig. 1.

Step 1 - Control Concern. The Control view shown in fig. 10 has already been described in detail in section 3.2 and is the same as the left part of fig. 4. The view shows a decomposition in agents and an environment.

Step 2C - Centralized. The goal of the first distribution schema (centralized) is to allocate all agents and the environment to a single server. To allow such allocation, we must provide remote access to the Ensor component which is situated on the AGV vehicle.

We start with defining two subsystems, shown in fig. 11, Controller and RemoteEnsor. They will be deployed on a single server and an AGV vehicle, respectively. Both have an interface for remote communication (connected to each other via Network). Furthermore, the Controller is connected to the WMS, RemoteEnsor to the Ensor system.

Next, we want to define the relation between the Centralized view and the Control view. The connector between Environment and Ensor in the Control view does not have appropriate support for remote communication yet. To solve this problem, we refine the EnsorCon in the next step.

Step 3C - EnsorConnectorInternal. To refine the EnsorConnector, we first define an auxiliary view and then compose this auxiliary view with the Control view.

Figure 12 shows an auxiliary view called `TensorConnectorInternal` defining the internal structure of the `TensorCon`. This structural view contains two subcomponents connected to each other by a connector: `TensorProxy` and `RemoteTensor`. `TensorProxy` will act as a proxy for `Tensor` from the perspective of the `Environment` component.

Step 4C - CentralizedAGVSystem. Finally, the Centralized view is composed with the ControlExtendedAlt1 view in fig. 13. We use both *mapping* and *unification* for this. The `EnsorProxy` component and all agents and the `Environment` component are mapped on the `Controller`. We unify the pairs of `RemoteEnsor`, `Ensor` and `WMS` components with each other. This results in the `CentralizedAGVSystem` view (shown at the right hand side of fig. 13). Notice that there is no explicit connector between `Environment` and `EnsorProxy`, or between `RemoteEnsor` and `Ensor`. At first glance this might seem strange, but the links between the respective components pass the border of `EnsorCon`. This is a result of describing a refinement of `EnsorCon`, meaning that the interfaces of the `EnsorCon` are delegated to internal components.

This wraps up the description of the first alternative for distribution, leading to a centralized system. The next step starts from the same Control view, but will build up the system aiming at a decentralized distribution schema (resulting in another, decentralized system).

Step 2D - Decentralized. 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. Again, we start by defining two subsystems in fig. 14, AGVControl and TransportBase, to deploy on the AGV vehicle and a server respectively. Both have an interface for remote communication (connected to each other via Network). Furthermore the TransportBase has a connection to the WMS, the AGVControl component to Ensor. Next, we want to define the relation between Decentralized view and the Control view. Here we have to solve the overlap/crosscut problem as described in fig. 6. To solve this, the Environment component is refined in the next step.

Step 3D - EnvironmentInternal. To refine the Environment component, we first define an auxiliary view and then compose this auxiliary view with the Control view. The left hand side of fig. 15 shows the auxiliary view defining the internal structure of the environment. The environment is split up in two local environment, AGVLocalEnvironment and TransportLocalEnvironment. These two local environments synchronize state take care of communication, and together they form a virtual representation of a single Environment component.

Defining the composition with this auxiliary view using the *refinement* relation is less trivial than the previous example. The Environment is split in two components, but three interfaces (perception, communication, action) are offered by both the AGVLocalEnvironment and the TransportLocalEnvironment, asking for a more complex interface mapping. In the composition specification we provide several alternative mappings and specify which connector uses which alternative. The right hand side of fig. 15 shows the composition specification. The composition results in the ControlExtendedAlt2 view that is used in the next step to compose with the Decentralized view.

Step 4D - DecentralizedAGVSystem. Finally, we can compose the system together to form a decentralized version of the AGVTS. Figure 16 contains the composition

Fig. 14. Decentralized view. This is the same view as the right hand side of fig.4, with resized elements to allow compacter presentation. Same graphical notation as fig. 1.

Fig. 15. (a) EnvironmentInternal structural view. Same graphical notation as fig. 1. (b) Specification to compose EnvironmentInternal with Control.

specification using the mapping and unification relations. This results in the DecentralizedAGVSystem view, looking exactly like fig. 3. This wraps up the description of the second alternative for distribution, leading to a decentralized system.

4.3 Discussion

Relations Between Structural Views Made Explicit. Using structural views to separate concerns and adding explicit compositions between structural views proves to be valuable from different perspectives. By using separate structural views, an architect can effectively separate several important concerns. With the extension to the xADL language, an architect can explicitly define the relations between different structural views. As discussed in the first section, this helps to improve the understandability and changeability of individual architectural concerns, such as in the distribution-specific scenario in the AGVTS architecture. It also becomes easier to compare two different distribution schemas for the same control view.

Notice that the resulting decentralized architecture looks the same as the original crosscutting design. The main difference is the process followed by the architect to design this structural view. The new decomposition has clearly separated the concerns and provided explicit composition specifications between the structural views.

Lessons learned. First, it became clear that using relations and compositions does not necessarily simplify the architectural description. There is a tradeoff between extensive&explicit specification and simplicity. Leaving assumptions and relations implicit is without doubt much simpler, requires less effort, and it may be easier to have an overall view of the system. But explicit specification does make the assumptions associated with individual concerns explicit, together with the relations between the structures. This makes it easier to manage and change the architectural description, clear advantages of the proposed solution that are essential for some architectures. One suggestion to improve the understanding of the overall view of the system is to include some re-

```

<archComposition id="DecentralizedAGVSystem">
  <description>Decentralized AGV agent system</description>
  <mapping>
    <target>
      <archStructure href="#Decentralized"/>
      <element href="#AGVControl"/>
    </target>
    <subStructure>
      <archStructure href="#ControlExtendedAlt2"/>
      <element href="#AGVAgent"/>
      <element href="#AGVLocalEnvironment"/>
    </subStructure>
    <interfaceMapping>
      <outerInterface href="#AGVControl.ensor"/>
      <innerInterface href="#AGVLocalEnvironment.ensor"/>
    </interfaceMapping>
    <interfaceMapping>
      <outerInterface href="#AGVControl.network"/>
      <innerInterface href="#AGVLocalEnvironment.network"/>
    </interfaceMapping>
  </mapping>
  <mapping>
    <target>
      <archStructure href="#Decentralized"/>
      <element href="#TransportBase"/>
    </target>
    <subStructure>
      <archStructure href="#ControlExtendedAlt2"/>
      <elements selectionStatement="Transport*" />
    </subStructure>
    <interfaceMapping>
      <outerInterface href="#TransportBase.network"/>
      <innerInterface href="#TransportLocalEnvironment.network"/>
    </interfaceMapping>
    <interfaceMapping>
      <outerInterface href="#TransportBase.wms"/>
      <innerInterface href="#TransportManagerAgent.wms"/>
    </interfaceMapping>
  </mapping>
</archComposition>

```

Fig. 16. Final composition in decentralized agent system for AGVTS. The refication of this composition is the same as fig. 3 and is repeated in fig. 14.

sults of the composition in the architectural documentation. Tool support is essential to make this feasible.

The second lesson we learned is that compositions are a natural extension to architectural description. Adding compositions to the xADL language was possible without touching the existing architectural concepts in the language. Additionally, there are already first hints in important architectural works that relations are important, as illustrated in the introduction. Yet, these relations receive only limited attention and are not explicitly embedded in the ADL. In this paper we go one step further by explicitly grouping the views and associated relations in compositions.

The idea behind the relations only relies on basic ADL elements, such as components, interfaces and connectors. As consequence, integration with for example the ACME ADL should be straightforward. Of course, the syntax needs to be adapted. Integration with ACME ADL is a matter of future work.

Limitations. The relations rely on specifying correspondence between interfaces (interface mapping). This restriction to one-to-one mapping might not work well in some cases. For example, one structural view could contain two interfaces which map on one interface in another structural view. Another example is the problem with the refinement in step 4D, in which an interface occurs on two subcomponents. The latter problem is solved by adding an alternative specification to refinement, but this might not always be possible. In some situations there will be no other choice but to refactor the interfaces of one of the structural view.

This paper focuses on structural views and composition of structural views. Other types of views are also possible and relations between these views need further investigation. We briefly discuss first insights into the matter, distinguishing between relations between the same type of views and relations between different types of views.

Fig. 17. Decentralized agent system for AGVTS, result after composition in fig. 16. We left out the Environment component (containing the AGVLocalEnvironment and the TransportLocalEnvironment) to make the figure more readable.

Firstly, we discuss relations between the same type of view. We expect that each type of view may need its own set of relations. That is because each type of view has its own set of basic elements and different semantics, like component and connector types as building blocks of a structural view or processes and communication channels as building blocks for a process view. Relations are described in terms of the basic building blocks and are only meaningful within the specific semantics of that particular view. This seems to confirm our expectations.

Secondly, we discuss relations between different types of views. We suspect that each pair of types of views may need its own set of relations, for the same reason as explained above. Several relations between different types of views are already supported in ADLs. For example, in an xADL instance view an architect can describe that this component is of a particular type described in a structural view.

Notice that for an architectural description, not every type of view needs direct relations with all other types of views. Some relations between views can be indirect, i.e. through another type of view. Describing meaningful relations in general in the context of heterogeneous representations is a challenging problem [36], because there exist a large quantity of types of views each with its own notation and semantics.

We proposed three relations. Each of the relations serves a specific purpose as outlined at the end of section 4.1. However, in other situations other relations might be needed. Other possible relations include relations to solve interferences between identifiers of software elements, relations to solve non-corresponding interfaces, etc. An important remark is that the total set of relations should stay rather small so that it remains easily comprehensible and usable for a software architect.

Further exploratory studies are required to assess the scalability of the approach. Currently, we only considered an excerpt from the architecture of the AGVTS. Further

studies are needed to see what the impact is in case of a large increase in views. Again, tool support will be essential to make the approach applicable in significantly larger architectural case studies.

5 Related Work

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 [1,2]. Additionally, 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 [33], can manifest early in design decompositions. The work in this paper fits within this trend, but has a strong focus on software architecture.

Some aspect oriented techniques for programming (e.g. HyperJ [43]) and requirements engineering (e.g. [11,34,41]) provide structural relations for enabling the separation of certain recurring crosscutting concerns. Surprisingly, composition mechanisms for modularizing architectural crosscutting structural views have been neglected in most of the available literature [4,13] focusing on discussing the interplay of aspects and ADLs. Batista and colleagues [4] 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 connectors and attachments can be extended to allow behavior-dependent aspect composition. In addition, existing AO ADLs [21,26,35,39,37,38] expose join points as nodes in a dynamic component or object call graph. Modularizing crosscutting concerns in structural views is not supported by this behavioral compositions mechanisms for software architectures, leading to another approach as advocated by this paper.

The presented approach is closely related to multi-dimensional separation of concerns (MDSOC) as introduced by Tarr et al. [43]. Each of the views corresponds to a hyperslice. The architectural composition we propose corresponds to a hypermodule. A hypermodule is a set of hyperslices, together with a composition rule that specifies how the hyperslices must be composed to form a single, new hyperslice that synthesizes and integrates their units. We defined a composition specification as an aggregation of several views and relations. Just as a hypermodule defines a new hyperslice, a composition specification defines a new view. The implication of such choice in our specific solution need further research, but at first sight such hierarchical composition can be beneficial for the scalability of the approach. Our relations correspond to the composition rules of MDSOC.

Quantification [19] over architectural elements is found in the selection statement for the elements specification. In the example, the expressions are rather limited (text and wildcards), but they can easily be extended.

In the field of software architecture, there is little support for relations between views. Rozanski and Woods [40] identify that quality concerns crosscut several views. The authors introduce architectural perspectives as complementary to views in the sense that they define a set of activities, tactics and guidelines to ensure that the view fulfils a quality. IEEE-Std-1471 [27] provides a strong conceptual model relating stakehold-

ers, 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.

Several approaches combine aspect orientation with multi-agent systems. Kendall et al. [30,31] describe the use of aspect orientation while designing and implementing role models for multi-agent systems (more recently continued in [8]). Contrary to this paper, the authors focus on implementation. In [23], Garcia et al. observe that several agent concerns such as autonomy, learning, and mobility crosscut each other and the basic functionality of an agent. The authors propose an aspect-oriented approach extrapolating implementation mechanisms to the architecture. More recently, Garcia et al. [22] identified crosscutting concerns for agent systems described in the ANote modeling language. Our work differs in its focus on relating architectural views. Boucké et al. [6] did some experiments to extrapolate the approach to the architectural level. The main conclusion was that the basic idea behind Theme/UML proved very interesting, but the language itself is less suited for architectural design. We introduce a language that is based on the same foundation (MDSOC, discussed before) but is embedded in an architectural description language.

6 Conclusion

The lack of explicit relations between structural views in an architectural description is a severe problem, leading to improper documentation of concerns that affect multiple structural views. Based on a concrete set of non-trivial examples, this paper illustrated that several relevant concerns crosscut the architecture structural views of an industrial automatic transportation system. Our systematic analysis of the architecture decompositions identified several challenges to relate structural views and we proposed an initial set of relations in xADL to cope with these challenges.

Through a first case study, this paper has illustrated the feasibility of composition and the concrete relations, namely refinement, mapping and unification. We have used these relations to refactor the architecture of a transportation system. One of the advantages is that distribution is clearly separated from control, making it easier to compare alternative distribution schemas. Within the limited setting of this paper, the results look promising. In future work, we plan to formally extend xADL by integrating the relations in the language definition and to investigate tool support to apply the compositions.

We should highlight that from our experience in previous case studies [18,25,33,10] the three relations and the composition seem to be recurrently demanding in architecture design processes. For instance, concerns such as error handling, distribution, and persistence, have been identified as crosscutting in the architecture of a Web-based information system [33,18], Eclipse CVS plugin [18,25], a reflective middleware system [10,9], and a traveller information system [18,25]. We have observed that in all these cases, advice-pointcut mechanisms, as supported by most AO ADLs, are not sufficient to cope with the separation and composition of such concerns. The reason is that issues like error detection, timeout control, and data synchronization [25] demand structural composition, and advice-pointcut is an inherently dynamic mechanism. Thus, we believe that the relations proposed in this paper are promising general-purpose

ADL additions to improve the stability of an architectural specification in the presence of change. We intend to perform further studies in order to confirm or refute our expectations.

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. Thanks to Danny Weyns, Dimitri Van Landuyt, the reviewers and the attendants of the Early Aspect workshop at AOSD 2007 for the discussions about the paper.

References

1. Araujo, J., Baniassad, E., Clements, P., Moreira, A., Rashid, A., Tekinerdogan, B.: Version 1 of the early aspects landscape report. Published on (2005), <http://www.early-aspects.net>
2. Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. *IEEE Softw.* 23(1), 61–70 (2006)
3. Bass, L., Clements, P., Kazman, R.: *Software Architectures in Practice*, 2nd edn. Addison-Wesley, Reading (2003)
4. Batista, T., Chavez, C., Garcia, A., SantAnna, C., Kulesza, U., Rashid, A., Filho, F.: Reflections on architectural connection: Seven issues on aspects and ADLs. In: *Workshop on Early Aspects held at ICSE 2006* (2006)
5. Boucké, N., Holvoet, T., Lefever, T., Sempels, R., Schelfhout, K., Weyns, D., Wielemans, J.: Applying the Architecture Tradeoff Analysis Method (ATAM) to an industrial multi-agent system application. Technical Report CW431, Dept. of Computer Science, KULeuven (2005)
6. Boucké, N., Weyns, D., Holvoet, T.: 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*, pp. 87–110. Net.ObjectDays, K.U.Leuven (2006)
7. Boucké, N., Weyns, D., Schelfhout, K., Holvoet, T.: Applying the ATAM to an architecture for decentralized control of a transportation system. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) *QoSA 2006. LNCS*, vol. 4214, Springer, Heidelberg (2006)
8. Cabri, G., Leonardi, L., Zambonelli, F.: Modeling role-based interactions for agents. In: *The Workshop on Agent-oriented methodologies at OOPSLA* (2002)
9. Cacho, N., Batista, T., Garcia, A., Sant’Anna, C., Blair, G.: Improving modularity of reflective middleware with aspect-oriented programming. In: *International Workshop on Software Engineering for Middleware (SEM 2006)* (2006)
10. Cacho, N., SantAnna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C.: Composing design patterns: A scalability study of aspect-oriented programming. In: *5th International Conference on Aspect-Oriented Software Development (AOSD 2006)* (2006)
11. Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-based composition for aspect-oriented requirements engineering. In: *Aspect Oriented Software Development conf* (2007)
12. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: *Documenting Software Architectures, Views and Beyond*. Addison Wesley, Reading (2003)
13. Cuesta, C., Romay, M., Fuente, P., Barrio-Solorzano, M.: Architectural aspects of architectural aspects. In: Morrison, R., Oquendo, F. (eds.) *EWSA 2005. LNCS*, vol. 3527, pp. 247–262. Springer, Heidelberg (2005)

14. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)* (2002)
15. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: 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)
16. Eclipse. Aspectj project, www.eclipse.org/aspectj/
17. Egemin. Egemin website, www.egemin.com
18. Filho, F., Cacho, N., Ferreira, R., Figueiredo, E., Garcia, A., Rubira, C.: Exceptions and Aspects: The Devil is in the Details. In: *Proc. Int. Conf. on Foundations on Software Engineering* (2006)
19. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA* (2000)
20. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): *Aspect-Oriented Software Development*. Addison-Wesley, Reading (2005)
21. Garcia, A., Chavez, C., Batista, T., Sant’anna, C., Kulesza, U., Rashid, A., Lucena, C.: On the modular representation of architectural aspects. In: *Proc. of the European Workshop on Software Architecture* (2006)
22. Garcia, A., Chavez, C., Choren, R.: Enhancing agent-oriented models with aspects. In: *Proceedings of the ACM Fifth International Joint Conference on Autonomous Agents & Multi Agent Systems* (2006)
23. Garcia, A., Kulesza, U., Lucena, C.: Aspectizing multi-agent systems: From architecture to implementation. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) *SELMAS 2004*. LNCS, vol. 3390, pp. 121–143. Springer, Heidelberg (2005)
24. Garlan, D., Monroe, R.T., Wile, D.: *ACME: Architectural description of component-based systems*. In: *Foundations of Component-Based Systems*, Cambridge University Press, Cambridge (2000)
25. Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant’Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the impact of aspectual decompositions on design stability: An empirical study. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 176–200. Springer, Heidelberg (2007)
26. Grundy, J.: Multi-perspective specification, design and implementation of components using aspects. *International Journal of Software Engineering and Knowledge Engineering* 10(6) (December 2000)
27. IEEE. Recommended practice for architectural description of software-intensive systems (ansi/ieee-std-1471) (September 2000)
28. ISR, Institute for Software Research. Archstudio 4.0 tool set for the xadl language, <http://www.isr.uci.edu/projects/archstudio/>
29. Jackson, M.A.: Some complexities in computer-based systems and their implications for system development. In: *Proceedings of CompEuro 1990*, IEEE Computer Society Press, Los Alamitos (1990)
30. Kendall, E.A.: Role model designs and implementations with aspect-oriented programming. In: *OOPSLA 1999. Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 353–369. ACM Press, New York (1999)
31. Kendall, E.A.: Role modelling for agent systems analysis, design and implementation. *IEEE Concurrency* 8, 34–41 (2000)
32. Kojarski, S., Lorenz, D.H.: Modeling aspect mechanisms: A top-down approach. In: *In Proc. Int. Conf. on Software Engineering*, pp. 212–221 (2006)

33. Kulesza, U., SantAnna, C., Garcia, A., Coelho, R., von Staa, A., Lucena, C.: Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In: *Proceedings of 9th International Conference on Software Maintenance - ICSM 2006* (2006)
34. Moreira, A., Rashid, A., Araujo, J.: Multi-dimensional separation of concerns in requirements engineering. In: *The 13th International Conference on Requirements Engineering (RE 2005)*, pp. 285–296 (2005)
35. Navasa, A., Pérez, M., Murillo, J., Hernández, J.: Aspect oriented software architecture: a structural perspective. In: *Workshop on Early Aspects, AOSD 2002* (2002)
36. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult! In: *ICSE 2003. Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA, pp. 676–681. IEEE Computer Society, Los Alamitos (2003)
37. Perez, J., Ramos, I., Jaen, J., Letelier, P., Navarro, E.: Prisma: Towards quality, aspect oriented and dynamic software architectures. In: *Int. Conf. On Quality Software* (2003)
38. Pessemier, N., Seinturier, L., Duchien, L.: Components, adl and aop: Towards a common approach. In: *Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04)* (2004)
39. Pinto, M., Fuentes, L., Troya, J.M.: A dynamic component and aspect-oriented platform. *Computing Journal* 48(4), 401–420 (2005)
40. Rozanski, N., Woods, E.: *Software Systems Architecture*. Addison-Wesley, Reading (2005)
41. Silva, L.: An aspect-oriented approach to model requirements. In: *Proc. of the Doctoral Consortium on Requirements Engineering (with RE conference)* (2005)
42. Software Engineering Institute. The sae aadl standard info site, <http://www.aadl.info/>
43. Tarr, P., Ossher, H., Harrison, W., Sutton, S.: N degrees of separation: Multi-dimensional separation of concerns. In: *Int. Conf. on Software Engineering*, pp. 107–119 (1999)
44. Weyns, D., Holvoet, T.: A reference architecture for situated multiagent systems. In: Weyns, D., Van Dyke Paruak, H., Michel, F. (eds.) *E4MAS 2006*. LNCS, vol. 4389, Springer, Heidelberg (2006)
45. Weyns, D., Parunak, H.V.D., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems: State-of-the-art and research challenges. In: Weyns, D., Parunak, H.V.D., Michel, F. (eds.) *E4MAS 2004*. LNCS (LNAI), vol. 3374, Springer, Heidelberg (2005)
46. Wooldridge, M.: *An introduction to Multiagent Systems*. John Wiley & Sons, Chichester (2002)