Experiences with Theme/UML for Architectural Design of a Multiagent System

Nelis Boucké, Danny Weyns, and Tom Holvoet

DistriNet, Department of Computer Science, KULeuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{nelis.boucke,danny.weyns,tom.holvoet}@cs.kuleuven.be

Abstract. In a recent R&D project, our research group developed an industrial AGV transportation system using a multiagent system (MAS). The software architecture of this system is modeled using several architectural views. In this paper, we study an alternative way of structuring of the architectural description using Theme/UML. Theme/UML is an aspect-oriented design approach that provides support for advanced separation of concerns.
Our goal is twofold. (1) We structure the architectural description based on important *architectural concerns* (such as coordination and distribution), instead of the current structure based on different viewtypes (module, component, deployment). The goal is to investigate the advantages and trade-offs of separating concerns in architectural design of MASs. (2) Currently, Theme/UML provides only support for detailed design. We aim to evaluate whether Theme/UML can be applied to architectural design.
The results of our experience are promising. The advantage of separating architectural concerns is that the resulting architectural description (1) is better aligned with the architectural concerns, and (2) facilitates the investigation of alternatives for those concerns. We illustrate this by changing the distribution schema of the AGV transportation system. A trade-off is that describing the concerns separately makes it more difficult to get an overall view on the system. Finally, the experiment shows that Theme/UML is not ready for architectural design. To improve support for architectural design of MASs, we propose several extensions to Theme/UML .

1 Introduction

In a recent R&D project called EMC^2 [1,2,3], our research group and Egemin[1] developed an innovative version of an AGV transportation system. The goal of this project was to investigate the feasibility of using a multiagent system (MAS) to improve the flexibility of the transportation system. The software architecture of this system is modeled with different architectural views, including module views, component and connector views, and deployment views [5].

To cope with the complexity of a large-scale MAS such as the AGV transportation system, a good separation of concerns is essential. Experience taught us that several concerns, specific to MAS (e.g. decomposition agents/environment, and coordination) and more general concerns (e.g. distribution, persistency, etc.), tend to crosscut each

[1] Egemin is a Belgian manufacturer of automated logistic systems for warehouses [4].

other in multiple views and components. We use the term *architectural concerns* to denote such concerns. An architectural concern is an area of interest or a focus of the software architect. Examples of architectural concerns for the AGV transportation system are the decomposition into agents and an environment, coordination, and distribution. In the project, a specific distribution scheme was chosen from the start. Distribution was not considered as a separate concern for which alternatives may be considered. As a result, distribution is scattered over various design models[2] of the software architecture. This makes it difficult to use an alternative distribution schema for the system.

In this paper, we experiment with Theme/UML [6] to investigate an alternative way of structuring the design models of the AGV transportation system. Theme/UML is an aspect-oriented design approach that provides support for advanced separation of concerns. The goal of our experiment is twofold. First, we aim to investigate the advantages and the trade-offs of separating architectural concerns in architectural design of MAS. Second, since a MAS structures a software system at the level of software architecture in the first place, we aim to evaluate whether Theme/UML—that currently only provides support for detailed design—can be extended for architectural design.

Overview. The remainder of this paper is structured as follows. Section 2 introduces the necessary background. Section 3 describes the motivations for using Theme/UML. In Sec. 4 we introduce Theme/UML and our architectural extensions. Section 5 illustrates the architectural description of the AGV transportation system structured according to the architectural concerns. In Sec. 6 we reflect on the design and the architectural extensions to Theme/UML. Section 7 discusses related work. Finally, we conclude in section 8.

2 Background

This section briefly introduces our position on MAS and software engineering, it explains the basic terminology of aspect-oriented software development (AOSD), and introduces the AGV transportation system.

MAS and software engineering. Today's distributed applications have to deal with highly dynamic operating conditions, such as dynamically changing workloads, continuous changes in availability of resources and services, etc. MASs are ascribed quality attributes such as flexibility, openness, and robustness that enables the systems to handle the dynamic operating conditions autonomously. Basically, a MAS structures a system in a number of autonomous entities, embedded in an environment, which cooperate in order to achieve the system goals [7].

With respect to software engineering and MAS, our position is that using MAS for large scale system development does not render typical software engineering techniques obsolete. On the contrary, we belief that MAS should be integrated with mainstream software engineering practice. Two common software engineering techniques form a recurring theme through this paper, namely software architecture and aspect-oriented design techniques.

[2] A view is an example of a design model. Other examples are module diagrams, interaction diagrams, statecharts, etc.

Software architecture covers the first design decisions that meet the essential quality requirements of the system. A common definition for software architecture is [8]: "the software architecture of a program or computing system is the structure or structures of the system, which comprise architectural elements, the externally visible properties of those components, and the relationships among them". It is generally accepted that architectures are too complex to be described in a simple one-dimensional fashion and must be described using several views [9]. Each architectural view shows particular types of elements and the corresponding relationships between them. Decomposing a system in agents and an environment and using specific mechanisms for coordination between the individual agents, determines the structure of the system and the fulfillment of the quality attributes. As such, a MAS in the first place structures a software system at the level of software architecture.

Aspect oriented software development aims to improve separation of concerns to cope with the complexity of large-scale software systems. Obviously, good separation of concerns is also important for MASs. During development there are always important concerns that are intermingled, thus hampering the development and maintenance of the application. Examples of concerns that a software architect has to conquer when developing MASs are coordination, agent mobility, learning, and autonomy (for a detailed discussion, see [10, pag. 175-246]). In Sect. 3, we give several examples of crosscutting concerns and point to the associated problems. Several approaches have been developed in the AOSD community to deal with crosscutting concerns. Most approaches however, are directed at the programming level or the level of detailed design. A good overview of existing AOSD approaches can be found in [11]. Recently, the role of software architecture in AOSD became subject of active research [12,13].

AOSD terminology. The terminology of this section is based on [11,14]. Since aspect orientation origins from programming level, some of the basic terminology closely resembles this level. Yet, we will provide some of the terms to show in the paper how this relates to architecture.

A *concern* is an area of interest or focus in a system, important for a stakeholder of the system. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to a software engineer. Some examples of concerns for a banking applications are the withdrawal of money, logging, and authentication.

Crosscutting of concerns means that concerns are both scattered and tangled with each other in a particular representation. *Scattering* means a single concerns is spread over multiple modules. *Tangling* means a single module contains parts of several concerns. For example: if a system is decomposed in modules and if the two concerns under consideration are logging and authentication, they are crosscutting if both concerns are scattered over multiple modules and tangled in several of these modules.

A *join point* is a well defined element of a language on which additional behavior can be attached. Examples of join points in the Java language are calling and executing a method. Notice that the notion of join point on the architectural level must use architectural language elements. Examples of join points for a components-oriented language are using a component service and changes in the observable behavior of a component. A *pointcut* defines a set of join points. Typically, a pointcut is specified using a list of join points or a regular expression over join points. An *advice* describes additional behavior to be executed on particular join points, thus defining a part of the crosscutting

- Module views
  1. Layered: The ATS (AGVs transportation system)
  2. Decomposition: **The ATS, Transport base, AGV Control system, Local virtual environment**, Transport agent, AGV agent, Decisions
  3. Uses: **Transport Base, AGV Control system**
  4. Generalization: **Agents**
- Component and connector views
  1. Shared date: **Agent, Local virtual environment, Protocol description**
  2. Process views: **Move action, Sending-Receive action, Background processes**
- Combined views: ObjectPlaces middleware, collision avoidance
- Deployment view: Deployment

Fig. 1. List of design models made for the AGV transportation system.

behavior. Advices are used together with a pointcut: the pointcut specifies at what join points the behavior of an advice has to be executed.

AGV transportation system. The application used in this paper is an Automatic Guided Vehicle (AGV) transportation system. This is an 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 that caries a computer that controls the vehicle. 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 completely drained.

The AGV transportation system interfaces with two external systems: Ensor and a Warehouse Management System. Ensor is a software system to control the AGV vehicle. More specifically, Ensor provides an interface with commands to steer the vehicle (e.g. move, turn). Internally, these commands are translated by Ensor into low-level commands to control the engines and sensors on the vehicle.

A Warehouse Management System (WMS) is a software system to manage products or raw materials in an industrial environment. Examples of responsibilities of such a system are transport management, stock management and accounting. The WMS interacts with the AGV transportation system by generating the transportation tasks. Additionally, the WMS keeps track of the ongoing tasks and is warned when a task starts and completes.

The remainder of the paper will mainly focus on three architectural concerns: (1) choosing the appropriate decomposition into agents and an environment; (2) handling the coordination between the agents; and (3) handling the distribution of the application.

3 Motivation to apply Theme/UML for architectural design of MASs

The motivation to apply Theme/UML for architectural design of MASs is twofold.

Our first motivation is to investigate whether Theme/UML is suitable for architectural design of MASs. Theme/UML is one of the best known aspect-oriented approaches for design, but experiences for architectural design are lacking. The goal is to

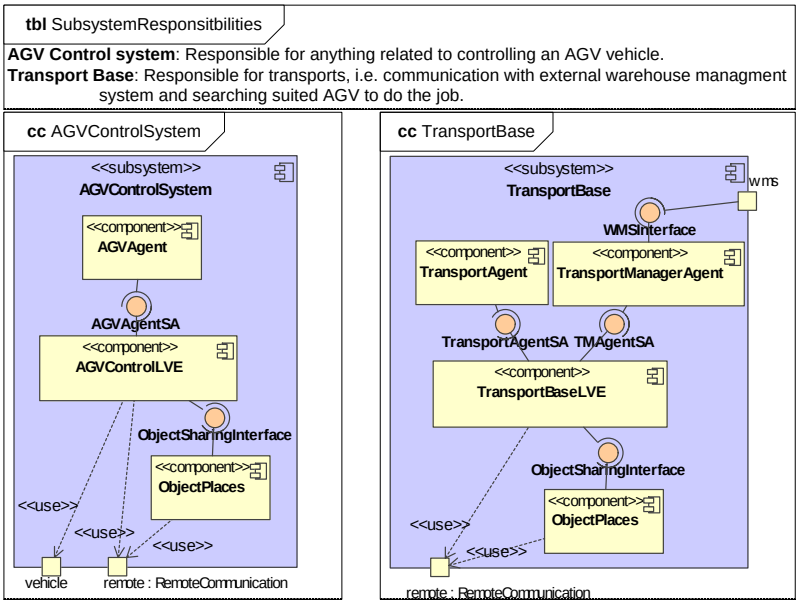

Fig. 2. Primary views of AGV transportation system design. Top: clarification table describing the responsibility of each subsystem. Left: C&C model showing the AGV-ControlSystem. Right: C&C model showing the TransportBase.

investigate whether Theme/UML can be extended for architectural design and to identify possible trade-offs and shortcomings of the approach with respect to architectural design.

Secondly, architects using MASs for large-scale systems must cope with multiple architectural concerns that tend to crosscut each other in the design models. Architectural concerns that are spread over multiple models are difficult to understand, and considering alternative solutions to a particular concerns is complex. The Theme/UML approach provides support for advanced separation of concerns in design and is specifically aimed to cope with crosscutting concerns. Our goal is to use Theme/UML to group the design models per concern, aiming for more coherent documentation and better support to investigate alternatives for the separated concerns.

3.1 Illustration of crosscutting concerns

In this section, we give two examples of crosscutting concerns in the existing architectural models of the AGV transportation system, and we explain the associated problems.

Coordination vs. decomposition agents/environment. The first example illustrates crosscutting between coordination amongst the agents and the decomposition into agents and the environment. Figure 1 lists the design models (in this case architectural views) of the architectural documentation of the AGV transportation system. We have annotated the list to illustrate how the two concerns are tackled in various design models.

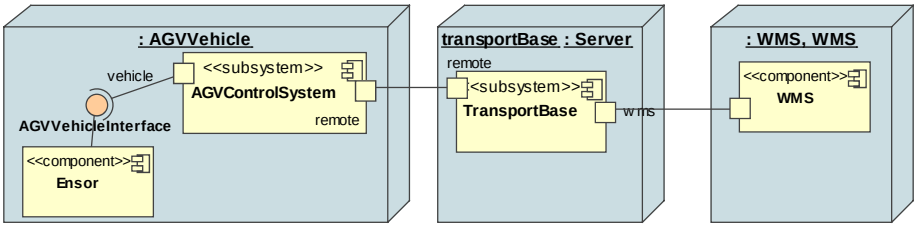

Fig. 3. Deployment view.

Design model names marked in bold include fragments of the agent/environment decomposition, underlined names include fragments of the coordination between the agents. Thus design models that are bold and underlined intermingle the definition of coordination with the decomposition into agents and the environment, illustrating that the two concerns crosscut each other in the design models.

Firstly, notice that the representation on which we identified crosscutting concerns are the *design models*, not the software components. As explained before, it is generally acknowledged that an architecture is described using several design models and choosing a good set of design models is very important for architectural design [9]. As advocated in [12], the design models themselves, rather than the software elements and relationships they show, form the dominant decomposition for an architecture.

Secondly, notice that the current organization of the architectural documentation is essentially based on the *types of architectural elements* like modules, components, and processes. The choice what to describe in a specific design model is left to the software architect. Since architectural concerns are often not made explicit, this choice is not made with the architectural concerns in mind [15].

When architectural concerns crosscut each other in several architectural models, an architect who wants to change one concern (e.g. coordination) must search within all design models, with few guidelines where to search. Even worse, because the relations between concerns are implicit, changing one concern in one design model may have unforseen effects on other concerns in other design models. As such, all design models must be reexamined and possible adapted to change something to a single architectural concern.

Distribution vs. decomposition agents/environment. As a second example of crosscutting concerns in the AGV transportation system, we consider the intermingling of the decomposition into agents/environment and the distribution of the system.

Figure 2 shows the design models of the AGV transportation system that correspond with the first three decomposition views of Fig. 1.

The system is decomposed in three types of agents (AGVAgent, TransportManagerAgent, and TransportAgent) and an environment.

The design of the environment assumes a specific distributed scheme. The assumptions are: (1) the deployment infrastructure that consists of mobile AGVs equipped with computers and a number of stationary computer systems; (2) each AGVAgent is part of a AGVControlSystem that is deployed on an AGV; and (3) the TransportAgents and a TransportManagerAgent are part of a TransportBase that is deployed on a stationary

computer system. Accordingly, the environment component is split up to provide environment services to the agents on each host. Therefore a local representation of the environment (Local Virtual Environment, LVE) and middleware to exchange information between hosts (ObjectPlaces component) are present on each computer system. Figure 3 illustrates the deployment of the subsystems on the hosts.

The three design models of Fig. 2 represent the basic decomposition on which the architecture is built. Yet, the design models clearly intermingle at least two different concerns (decomposition agent/environment and distribution). What belongs to which concern is not made explicit, only the agent and environment components for the assumed distribution schema are visible. Since distribution and the decomposition into agents and an environment are not considered as separate concerns, it becomes difficult to compare alternative solutions, or to change one of the concerns later on.

Currently Egemin considers to introduce the MAS solution step-by-step, starting with a centralized server that contains all the agents. In such a setup, remote communication would only be needed to communicate with Ensor (the vehicle control software). However, the original MAS design assumes as particular distribution schema, considering alternatives for distribution was not an upfront requirement. A consequence is that the assumptions about distribution are implicitly embedded in the design models, and that it is unclear what exactly needs to be changed. For example, in a centralized setting, it makes no sense to split up the environment. But, since all other views are based on the primary decomposition with LVE components, changing the architecture is quite difficult. All views assuming the use of the LVE need to be changed, implying that nearly all design diagrams need to be adapted.

The basic reason why this change is so difficult is that the design models are not structured according to the architectural concerns. Structuring the software according to important architectural concerns has the advantage that it becomes easier to change the architecture and to consider alternative solutions. In Sec. 5, we show that reorganizing the current architecture based on architectural concerns allows to investigate different distribution schemas for the AGV transportation system.

Reflection. In both examples, the problems with crosscutting concerns stem from the organization of the design models. As noted before, the architectural models are organized according to *types of architectural elements*, grouping modules together, grouping components together, etc. Theme/UML promotes a way of structuring according to architectural concerns, grouping all architectural elements belonging to a particular concern together. After all, each architectural concern determines a part of the structure, as such it seems logical to describe each concern as a separate block.

Notice that we are not questioning the value of the current design. The architects designed the system for flexibility and performance, using the guidelines for organizing architectural documentation of Clements et al. [9]. We only argue that a reorganization of the design models, with minimal changes to the architecture, could largely improve the coherence of the architecture documentation and easily allow to investigate alternative solutions for various concerns. In Sec. 5 we illustrate the design of the AGV transportation system restructured according to the architectural concerns and show that comparing different distribution schemas or changing the distribution schema, is easier. But first we introduce Theme/UML and a number of architectural extensions needed to describe the architecture of the AGV transportation system.

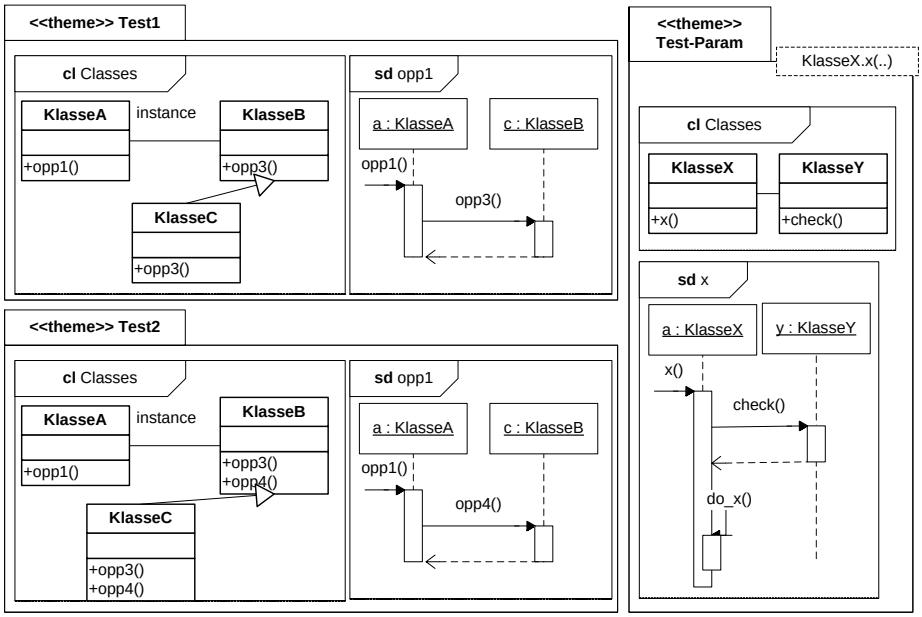

Fig. 4. Three example themes: Test1, Test2 and Test-Param.

4 Theme/UML and architectural extensions

Theme/UML is an aspect oriented design approach, introduced by Clarke et al. [16,6]. Historically, the approach originates from the work on subject oriented programming and design [17,18] and multi-dimensional separation of concerns [19]. Currently, it is one of the best known and documented aspect-oriented design approaches. The Theme/UML languages is an extension to UML[3].

The essential idea behind the Theme/UML approach is to structure the design (and the corresponding design models) according to the important concerns. In this section we briefly explain the Theme/UML essentials, together with extensions we have introduced for architectural design.

A theme. A *theme* describes a part of the design using several design models (e.g. class diagrams or sequence diagrams), where only the portions relevant to a particular concern are shown. Essentially, a theme groups several design models together that describe a single concern.

Themes are graphically represented by a UML package with a <<theme>> stereotype. Yet, there is a significant difference between themes and packages. Themes must be declaratively complete (meaning all elements used in the theme must be defined within the theme) and are always described independently from other themes (there is no “include” or “import” statement like for packages). Relations and dependencies between themes are specified during the composition.

[3] Originally defined as extension to UML 1.3 in [20], later used in UML 1.4 and UML 2.0.

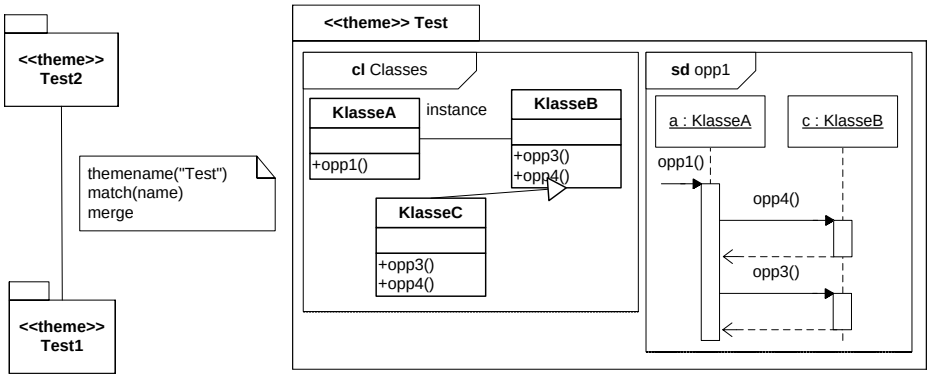

Fig. 5. Composing Test1 and Test2 together in the composed theme Test.

Themes can be composed together to form new themes. Themes defined by a composition of other themes are called composed themes, the other ones are called basic themes. For example, Fig. 4 contains three basic themes (`Test1`, `Test2` and `Test-Param`), Fig. 5 contains the composed theme `Test` and Fig. 6 contains the composed theme `System`.

In the original Theme/UML approach, design models are either class or sequence diagrams. We consider design models in a broader sense, a design model can be: graphical (e.g. UML class diagram) or textual (e.g. a table with responsibilities for each component or a formal description of the behavior); a general design model (e.g. an architectural view or an UML diagram) or a domain specific model (e.g. Agent UML (AUML [21])). For clarity every design model is named and embed in a frame. Embedding all design models in frames is not standard in UML or Theme/UML, both approaches only embed sequence diagrams in a frame. But we have used the frames to allow a better visual separation between models within a theme, especially in case of multiple design models. The types of design models are sequence diagrams (sd) and class diagrams (cl); and we have added state diagrams (st), clarification tables (tbl), C&C diagrams (cc) and deployment diagrams (dpl). The type of the design diagram is visible in the upper-left corner of the frame. For example in Fig. 4, all themes contain one class and one sequence diagram.

Parameterized themes. A theme can optionally contain *parameters*, this defines patterns that can be instantiated later on [22]. The parameters are specified in a dashed box at the upper right corner of the theme. Themes with parameters can be instantiated multiple times and a parameter can be bound to multiple values. A parameterized theme describes crosscutting behavior.

For example in Fig. 4, the `Test-Param` theme contains one parameter, being the `KlasseX.x(..)` method (the dots mean that the parameters are unspecified). The sequence diagram `behavior` specifies that when the operation `x` is called (which is a parameter), the sequence of the parameterized method must be changed by first calling the `check` operation and then the behavior of the original method. As such, the `Test-Param` theme adds the check behavior to the methods where `KlasseX.x()`

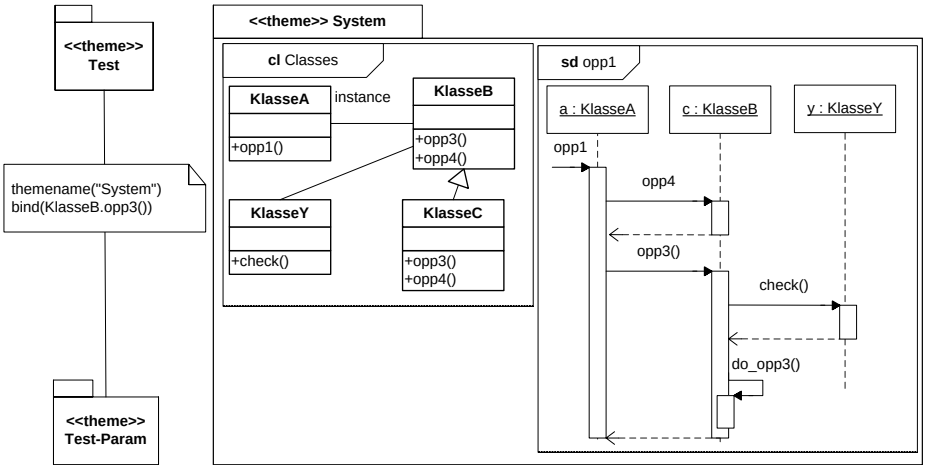

Fig. 6. Composing Test and Test-Param together in the composed theme System.

is bound to. The composition is described in details in Fig. 6 and the result is explained in details when introducing the bind tag.

Composition of themes. Once the concerns are designed in separate themes, the themes must be integrated to understand the system as a whole.

Theme composition is done in a theme composition diagram. Relations are represented by lines connecting the themes. Every relation is annotated with a UML note containing composition tags (additional information about how the concerns should be related / composed). Composition tags must be interpreted from top to bottom. Typically, a composition relationship contains tags identifying corresponding design elements in the related models and tags specifying how the corresponding elements must be integrated. For parameterized themes, the composition relationship contains binding tags to bind the parameters to concrete values. Possible tags are:

themename(”x”): Specifies the name of a composite theme. In Fig. 5 the composed theme `Test` describes the composition between theme `Test1` and `Test2`. In Fig. 6 the `Test` theme is further composed with the `Test-Param` theme, this new composed theme is called `System`.

match(name): Matches the elements of different themes based on their name. The match relations only applies for elements of the same type. The match relations is used in Fig. 5 to match elements from theme `Test1` with elements form theme `Test2`. In the example, all classes match with each other. The operation `opp1` is defined in both themes, but but the behavior specified in the themes is different and must be integrated (see further).

map(expression,componentType): Maps the component types in expression on the type componentType. The expression can either be a single type (e.g. map(typeA,typeB)), a list of types (e.g. map({typeA1,typeA2},typeB) or a regular expression on types (e.g. typeA+ stands for typeA and all subtypes, typeA*

stands for all types of which their name starts with typeA). The map tag is not part of standard Theme/UML.

merge: Merge matching elements. For each type of model and type of element, one have to specify what happens in case of a merge. An example can be found in Fig. 5. The figure contains both the composition specification (left) and the reification of this specification (at the right). As can be seen from the figure, both the class definitions and behavior have been merged.

override(themeA,themeB): Overrides matching elements from themeA with elements from themeB. This tag is part of standard Theme/UML, but we have not used it.

bind(params): Binds concrete values to parameters. "params" stands for a list of concrete values to fill in. For example, Fig. 6 binds theme `Test-Param` to the theme `Test`. More concretely, the composition tag bind(KlasseB.opp3()) specifies to bind `KlasseB` to `KlasseX` and `opp3()` to `x()`. The result of the binding is that the behavior specified in the parameterized theme (`Test-Param`) is inserted in the behavior of the theme it is composed with (`Test`). The right part of Fig. 6 is a reification of the binding.

Finally, we introduce *interaction refinements*. An interaction refinement is a relation between two design models (not themes!). It is not part of standard UML or Theme/UML. Figure 11 contains an example of the interaction refinement. Interaction refinement is indicated by a forked arrow between design models. Above the arrow is the description of the original interaction, below the arrow is a description on how the interaction must be refined. Using AOSD terminology: the part above the arrow (a situation description) together the bind tags roughly corresponds to a pointcut and the part below the arrow roughly corresponds to an advice.

Overview of our architectural extensions to Theme/UML Firstly, we integrated several new types of design models, namely state diagram, clarification table, deployment diagram, AUML sequence diagram.

Secondly, we introduced support for refinement, both for components and interaction. Refinement on the level of components is supported by the map tag that can be added to the composition specification, mapping several component of a theme on a single component of another theme. Refinement on the level of interaction is added by the interaction refinement relations between design models.

Finally, we used a slightly different notation to specify the composition relation between themes. All composition tags are specified in a single composition specification –a UML note attached to the composition– and must be interpreted from top to bottom.

5 Design of the AGV transportation system

In this section we illustrate the design of the AGV transportation system restructured according to the concerns, and we show that changing the distribution schema and comparing different distribution schemas is easier with the new design[4].

[4] From now on we refer to the design of the motivation section (and associated previous work) as the original design, the design of this section is called the new design.

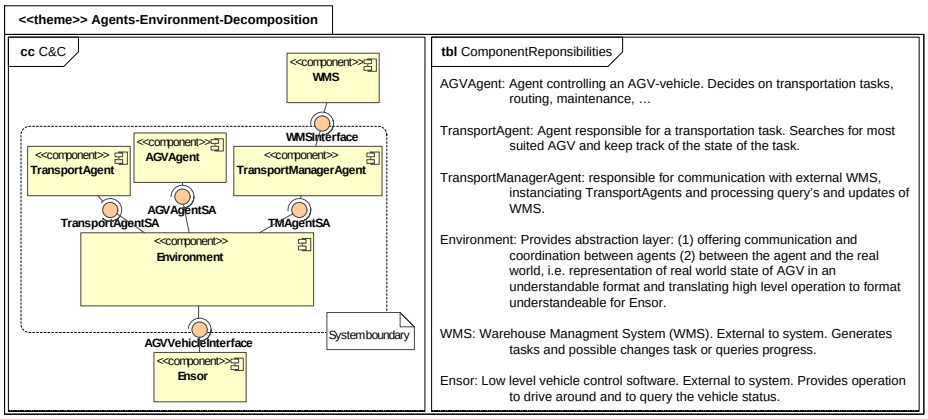

Fig. 7. Theme describing the decomposition into agents and environment.

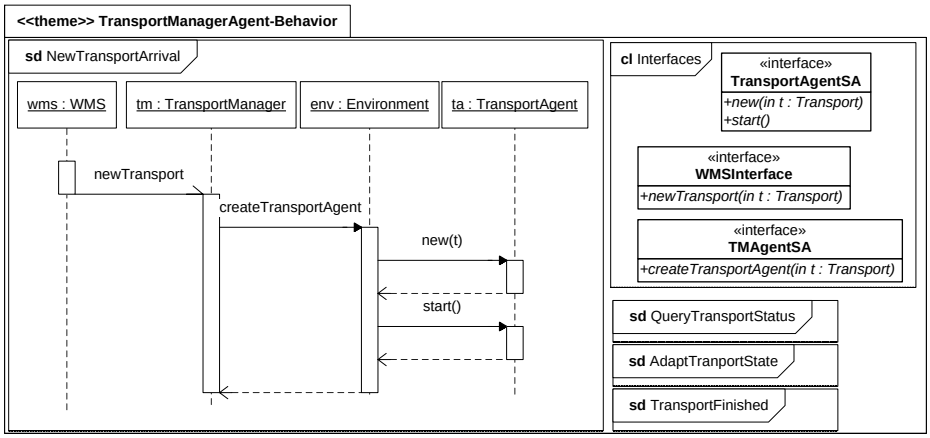

Fig. 8. Theme grouping the behavior of the TransportManagerAgent.

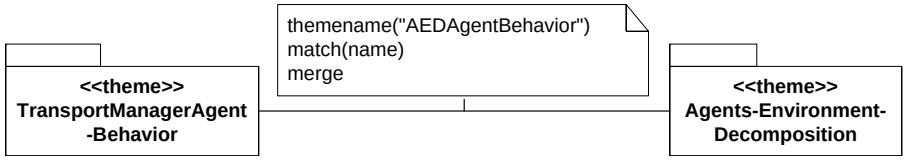

Fig. 9. Composition of Agent-Environment-Decomposition theme (Fig. 7) and TransportManagerAgent-Behavior theme (Fig. 8).

In Sec. 5.1 we start from two important architectural concerns to define basic themes, the decomposition into agent and an environment and the behavior of the individual agents. In Sec. 5.2, the coordination protocol between the agents is introduced. In Sec. 5.3 we add distribution into the design, describing two deployment alternatives: centralized and decentralized deployment of agents. Finally, we reflect on the design and the architectural extensions of Theme/UML in Sec. 6.

5.1 Decomposition of agents and individual behavior

This section starts from two important architectural concerns: the decomposition into agent and an environment (`Agents-Environment-Decomposition` theme) and the behavior of individual agents (we only elaborate on the behavior of the TransportManagerAgent in the `TransportManagerAgent-Behavior` theme). We describe both themes in more detail and describe how they can be composed.

The `Agent-Environment-Decomposition` theme defines the decomposition into agents and environment, and the responsibilities of the component types. Figure 7 shows the theme and its two design models. The first model, called `C&C`, contains a component/connector view showing the component types, the connectors and the respective interfaces. Two components are external to the system, Ensor and WMS. We decided to include them into this decomposition view to illustrate how and where they interact with the system. The second model, a clarification table called `ComponentResponsibilities`, lists the main responsibilities of the individual components. Notice that the relations between elements in different design models within a single theme are implicit, based on the names and types of the elements. For example, the TransportAgent appears in both models (`C&C` and `ComponentResponsibilities`) and represents the same component.

The `TransportManagerAgent-Behavior` theme in Fig. 8 describes the behavior of the TransportManagerAgent and the respective interaction with the environment. Only the arrival of a new task together with the creation of a TransportAgent is detailed, other possible interactions are represented by the empty model frames. We decided to provide a separate theme to describe the behavior of each agent, separating the behavior description from the decomposition.

The two themes of this section are composed in the `AEDAgentBehavior` theme in Fig. 9. The composition tags specify that matching is based on names and that merge integration is used to compose the themes together. In this particular example, there is no real merging because all models are of a different type. The composition groups the models together and because of the implicit relations between the design models within the new theme, the `TransportManagerAgent-Behavior` theme adds interface specifications and behavior to the `Agent-Environment-Decomposition` theme.

5.2 Adding coordination

In this paper we focus on a single subproblem for coordination: the allocation of tasks to AGVs. Since there is no central controlling entity, the agents have to agree amongst each other which AGV performs what transport. To solve this problem, we have used the well known Contract Net (CNET) protocol. The CNET protocol is originally proposed by Smith et al. [23]. We used the FIPA-CNET protocol described in the FIPA standard [24], slightly extending the original protocol.

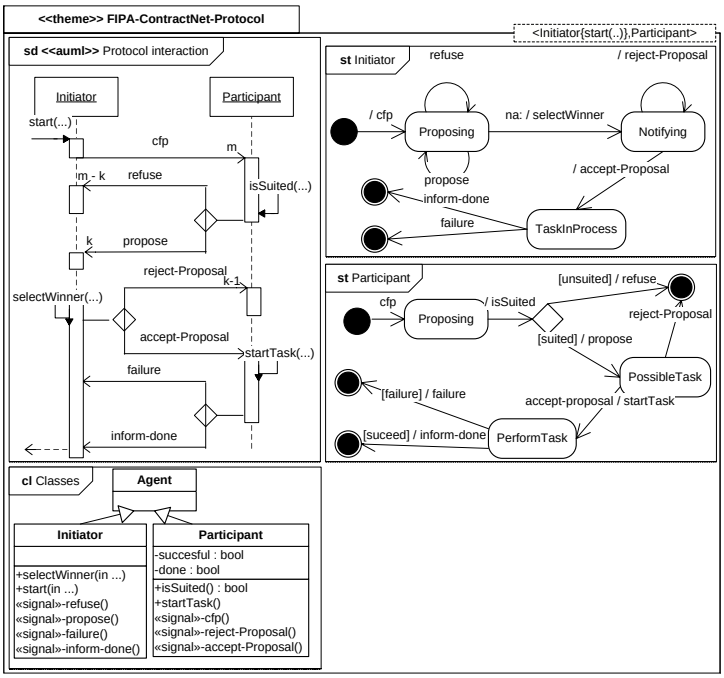

Fig. 10. Theme describing the FIPA Contract Net protocol.

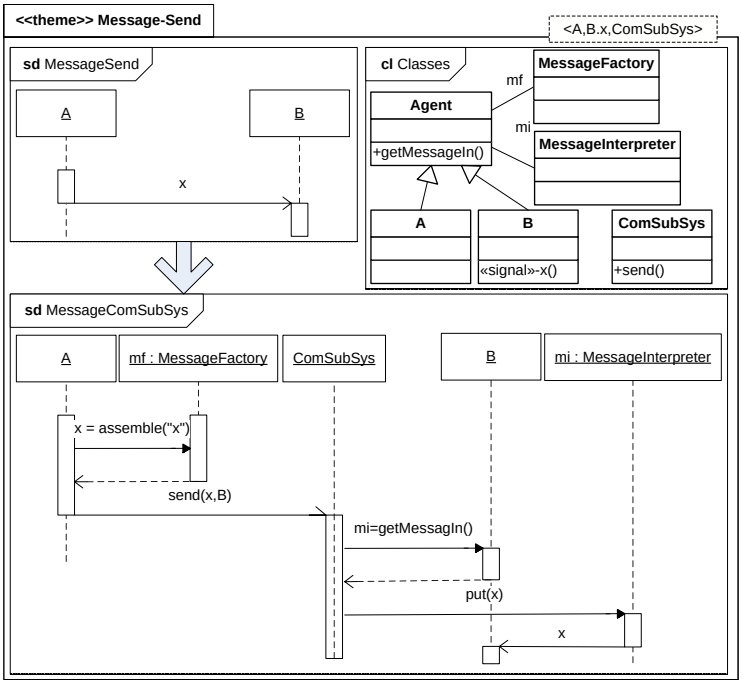

Fig. 11. Theme describing the handling of a message between agent A and agent B.

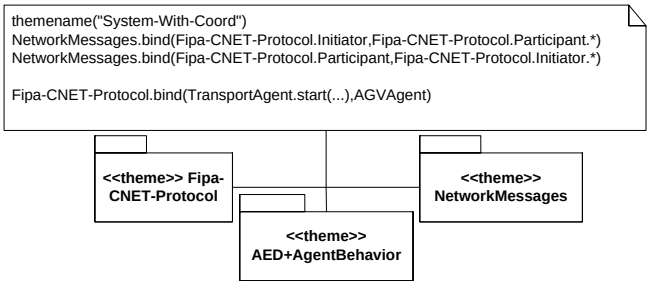

Fig. 12. Composition of CNET coordination and agents/environment.

Figure 10 describes the CNET protocol as a parameterized theme, called `Fipa-ContractNet-Protocol`, independent of a particular application. The parameters are specified when binding the theme with a particular application, e.g. in Fig. 12 the CNET protocol is bound to the agents of the AGV transportation system.

The CNET protocol description contains four models. The ProtocolSequence model describes the interaction between the agents, using the AUML language [21]. Notice the difference between operations calls (black arrow head) and sending messages. This diagram does not specify how these messages are sent. The Initiator and the Participant design models describe the internal states and the transitions between these states for the respective agents. The annotations of the transitions are '`event [condition] / action`'. The 'na:' in the transition between Proposing and Notifying stands for an event triggered by a timer. Finally, the Classes model describes the interfaces of the participating classes.

Before being able to bind the CNET protocol to the decomposition into agents and an environment in the AGV transportation system, we must solve a mismatch between them. At the one hand, the CNET protocol is described as an interaction between agents, abstracting away how exactly this interaction takes place. At the other hand, in the decomposition into agents/environment all interactions are mediated by the environment. To solve the mismatch we defined a new theme `Message-Send` to describe message sending between two agents, in Fig. 11. The `Message-Send` theme contains an interaction refinement indicated by an forked arrow. The `Message-Send` theme describes that each signal x between types A and B (i.e. the interaction matching model MessageSend) is replaced by assembling the appropriate message, sending it through the communication subsystem and interpretation and signalling by the MessageInterpreter. We still abstract from how messages are handled within the environment.

Finally, we come to the binding of the CNET protocol, sending of messages and the agents/environment decomposition, illustrated in Fig. 12. For clarity, we add a prefix with the theme name to the elements used in the binding. As last illustration, we partially instantiated the composition of coordination and the decomposition into agents/environment in Fig. 13.

5.3 Adding distribution

To add distribution, we first considered the distributed infrastructure on which the AGV transportation system has to be deployed (Fig. 14). The architectural decisions on how

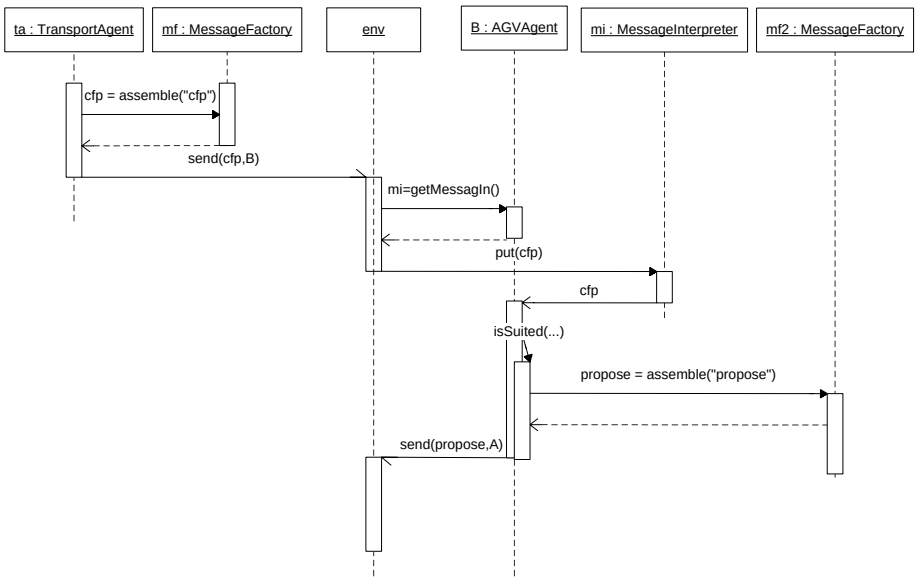

Fig. 13. Partial instantiation of composing between coordination and the decomposition into agents/environment.

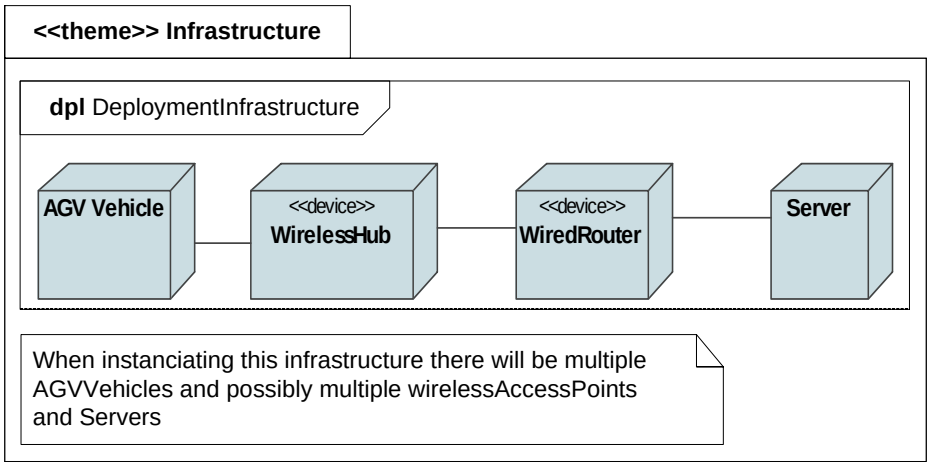

Fig. 14. Partial instantiation of composing between coordination and the decomposition into agents/environment.

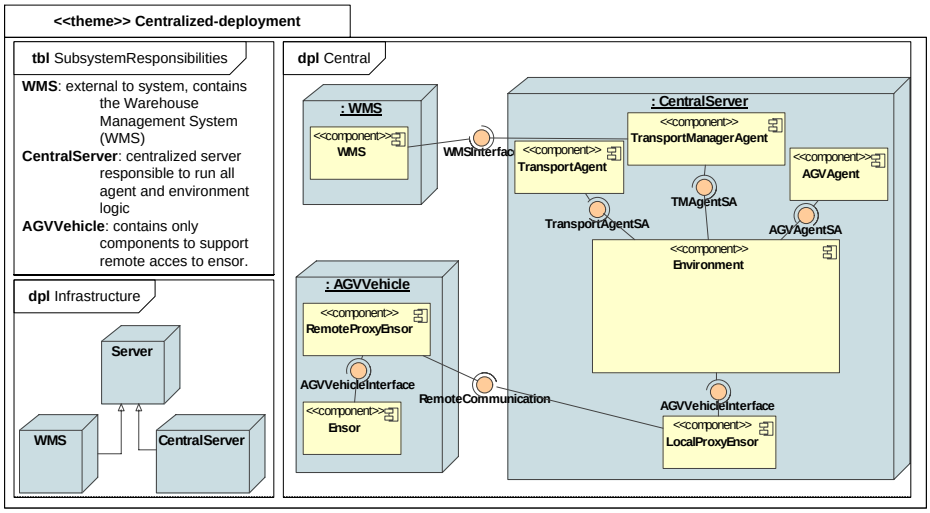

Fig. 15. Centralized deployment.

to deploy the system on this infrastructure are important because of significant influence on the qualities and the structure of the system. We describe two deployment alternatives: centralized deployment —with all agents on a single server—, and decentralized deployment —with the AGV agents located on the AGV vehicles—.

Centralized deployment The idea behind the centralized deployment schema is to keep all computation on a centralized server. This deployment schema is described in Fig. 15. The system has two subsystems: an AGVVehicle subsystem and the CentralServer subsystem. In this case, all agents are deployed on a central server subsystem and the AGV vehicle subsystem only contains the vehicle control software and a remote interface to handle communication with the environment component on the central server subsystem.

Notice that central deployment is easy, only the communication with the remote vehicle software needs to be added. The LocalProxyEnsor could also contain some caching mechanism to speed up the communication between the Environment and Ensor. Figure 16 describes the relations between the `Infrastructure`, the `Centralized-Deployment` and the `System-With-Coord` theme.

Decentralized deployment The idea behind the second deployment schema is to distribute the computation over the AGV vehicles, described in Fig. 17.

The decentralized deployment schema has also two subsystems: the AGV Control subsystem and the Transport Base subsystem. The AGV Agents are deployed on the first subsystem, a transport manager and all transport agents are deployed on the second subsystem. Because the agents are spread over multiple computer systems, the environment needs to be split up into several components and additional support is needed for communication and state maintenance between the different parts of the environment (provided by an in-house developed middleware called ObjectPlaces [25,26]).

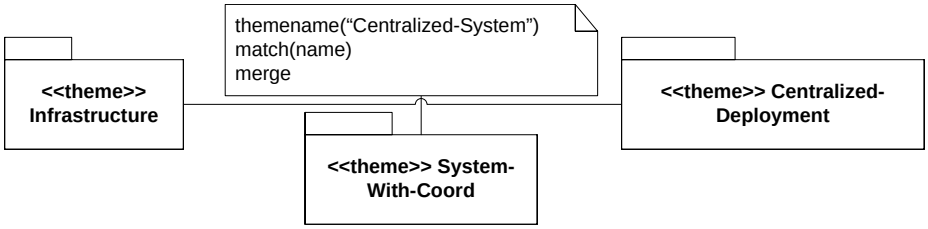

Fig. 16. Composition of infrastructure, centralized deployment and the remainder of the system.

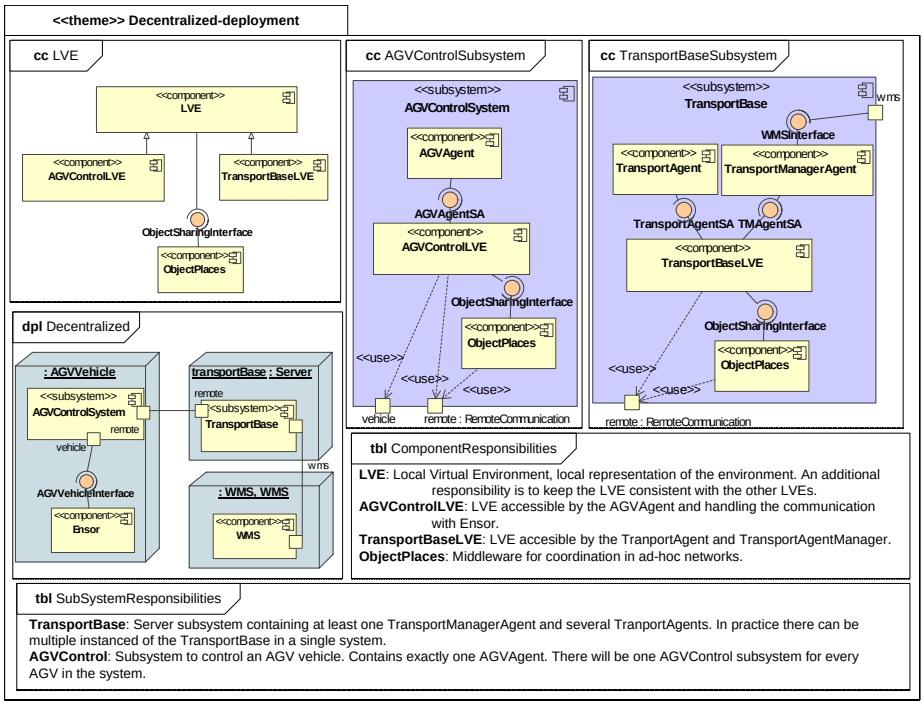

Fig. 17. Decentralized deployment and associated refinements of design.

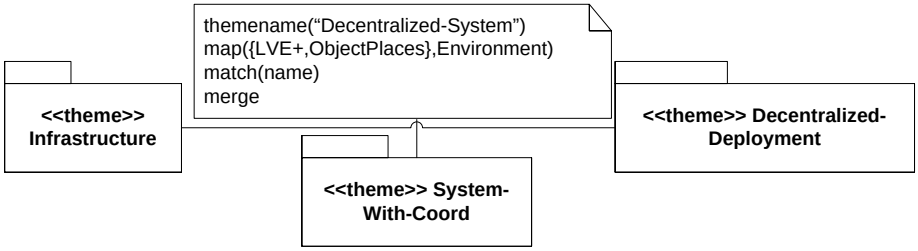

Fig. 18. Composition of infrastructure, decentralized deployment and the remainder of the system.

The Decentralized-Deployment theme contains five design models. Two clarification tables explain the responsibilities of newly introduced subsystems and components. The AGVControlSubsystem and the TransportBaseSubsystem C&C diagram explain the internal structure of the respective subsystems. The LVE C&C diagram shows the generalization relation between the LVE and the specialized AGVControlLVE and TransportBaseLVE. The final design model, the Decentralized deployment diagram, describes the deployment of the system.
Notice that the design diagrams of Fig. 17 (new design) are similar to the diagrams of Fig. 7 (original design). As we have explained before, the first design models in the original design are the result of relating the decomposition into agents and an environment and the deployment infrastructure, the same is true for the models in Fig. 17. Additionally, we did not alter the design, we only restructured it. The new descripstion also contains separate models for the decomposition into agents and an environment and a separate model describing the infrastructure. These models are lost in the original documentation.

Finally, we describe the relation between the Infrastructure, Decentralized-Deployment and the SystemWithCoordination theme in Fig. 18. The composition maps the LVE+ (LVE and all subtypes) and Objectplaces on the environment. For the remainder we use straightforward matching by name and merge integration.

6 Discussion

6.1 Design of the AGV transportation system

Using Theme/UML for the architectural design of the AGV transportation system proved to be a valuable experience. Restructuring the design according to the important architectural concerns makes comparing alternative solutions or changing the architecture easier. Section 5 shows that the new design (1) makes it easier to compare two different deployment schemas for the same decomposition into agents/environment, and (2) allows to change the deployment of the software without any change to the design for coordination.

It is important to notice that reifying the composition of the new design will yield the same system as in the original design. The difference is that the architectural description is structured to align with the architectural concerns. Since each concern determines a specific part of the architectural structure, describing this structure as a single block makes it easier to change the concern. For example, since all decisions for distribution are described in a single theme, it is easier to change the distribution.

Critical notes. Obviously, the advantages of separating concerns only applies to the concerns that are explicitly considered during architectural design. As such, choosing the appropriate architectural concerns is very important.

The composition of different parts of the design is not always easy. Assuming that all composition relations can be defined using a few composition tags would be naive. For example, we encountered a mismatch between the structures of the FIPA-ContractNet-Protocol and the Agent-Environment-Decomposition themes which need to be composed (see Fig. 10 and Fig. 7 respectively). To solve this

problem, we first build the additional `System-With-Coord` theme that defines how the different structures should be matched with each other. Afterwards, we composed the mismatching themes with the additional theme, forming a solution to the composition.

A drawback of the new design is that it is more difficult to have an overall view on the system. While the structures of individual architectural concerns is much clearer, reification of some compositions will be needed to to get an overall view of the system. Adding reified design models in the architectural documentation helps to improve the understanding of the system as a whole.

In our experiment, only a small part of the design of the AGV transportation system has been considered. How scalable the approach is in case of a large increase in concerns and themes needs furter study.

Each theme describes a separated concern, however, a theme provides no encapsulation. All parts of the structure and behavior of a theme can always be overriden or changed by another theme. For example, in Fig. 4 there is no direct composition relations between `Test2` and `Test-Param`, but still `Test-Param` changes the behavior specified in `Test2`. The change happens because Test-Param is composed with `Test`, being a composition of `Test1` and `Test2`. Clearly, this can lead to unexpected interferences between concerns. As a consequence, to determine if the behavior has changed or is overridden one must study the relations of the theme and all composite themes containing this theme.

6.2 Theme/UML

We point out a number of limitations of Theme/UML for architectural design of MAS, and we explain the extensions we have introduced to deal with these limitations.

There is no support to integrate new types of design models. As explained in section 4, the UML specification contains several types of diagrams related to detailed design. MAS architects however, use design models specific for architecture (such as the set of diagrams used by Clements et al. in [9]), and MAS-specific design models (such as AUML). The integration of new types of design models in this paper shows that new models can be integrated relatively easily. Still, disciplined guidelines to integrate new types of models are essential to extend Theme/UML for architectural design.

Theme/UML has no support for gradual refinement of design models, however this is essential for architectural design. To allow refinement of components, we added the map tag to the Theme/UML composition. A map tag expresses the refinement of a component in one theme in several subcomponents belonging to other themes. To support refinement of interactions, we introduced an interaction refinement relation that is inspired by the work of Atkinson et al. [27,28]. Interaction refinement relations, together with the parametrization of Theme/UML, provides a powerful way of refining interactions between components.

Theme/UML only supports execution joinpoints. Figure 19 illustrates the difference between call joinpoints and execution joinpoints. Joinpoints are represented by a black disc. This is a severe limitation, both for detailed and for architectural design.

There is a lack of tool support for Theme/UML. A designer using Theme/UML must use a standard UML drawing tool and must do every reification of a composition by hand, a cumbersome process. A tool that includes support for automatic reification of compositions would be most helpful.

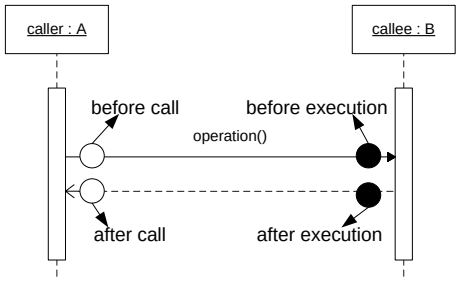

Fig. 19. Composition of infrastructure, decentralized deployment and the remainder of the system.

7 Related work

Software architecture in MAS. Several researchers use the notion of software architecture and architectural design (based on [8,9,29]) to develop MASs. Shehory [30] considers MAS from the perspective of architectural styles, to reasons about the qualities that are typically attributed to the MAS styles. PROSA [31] offers a reference architecture for coordination in manufacturing control. Our research group defined a reference architecture for situated MAS [32] and developed an industrial AGV transportation system using MAS. The software architecture of this system is modeled with different architectural views and structured according to different viewtypes (module, component, deployment). The approach promoted in this paper differs from previous approaches, since it advocates a way of structuring the architectural models based on important *architectural concerns*.

Aspect orientation and MAS. Aspect orientation is rarely used to develop MAS. Kendall describes the use of aspect orientation in the context of MAS in [33,34] (more recently continued in [35]). The approach uses aspect-oriented programming to design and implement role models.
In [36], 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 to develop agent architectures, using two types of interfaces: regular and crosscutting interfaces. A crosscutting interface specifies when and how an aspectual component crosscuts other components. The authors claim that the proposed approach provides a clean separation between the agent's basic functionality and the crosscutting agent properties. Unfortunately, a precise semantics of the crosscutting interface is lacking. The differences with the work in this paper are: (1) we focus on MASs as a whole, identifying concerns that span several agents; (2) we use the well known aspect-oriented design language Theme/UML; and (3) we identify crosscutting and separation of concerns on the level of design models, not components.
More recent, Garcia et al. [37] identifies crosscutting concerns for agent systems described in the ANote modeling language. ANote defines several domain specific views, e.g. an agent view, a goal view and a scenario view. Both in the goal and scenario view crosscutting concerns have been identified and a new notation is provided to cope with this crosscutting. The main differences with the work in this paper are: (1) the authors

used a domain specific language ANote, while we use several types of design models based on UML; and (2) the authors provide a specific extension to the domain specific language, while we use the model separation capabilities of Theme/UML.

Architectural design and aspect orientation. Next to Theme/UML there are several aspect-oriented design languages. A good survey on design approaches (both architectural and detailed design) can be found in [38,39]. We briefly introduce two approaches closely related to the work in this paper.

Atkinson and Kühne [28] propose architectural stratification to combine the strengths of component-based frameworks and model-driven architectures to support AOSD. The approach is about gradual refinement of architectural structures base on architectural concerns, introducing interaction refinement as a relations between different design models (in this case architectural views). Our notion of interaction refinement is based on this work.

Katara and Katz [40] observe that incremental design of aspects has been neglected and that cooperation or interference between aspects should be made clear at the design level. The work is strongly related to Theme/UML, since it provides a similar extension to UML. The concern diagram of that paper is similar to our notion of concern composition diagram. The main difference is that Katara et al. focusses on how aspects can be combined to treat different concerns of a system and possible interactions between concerns, while the work in this paper and Theme/UML only use the diagram to describe the composition of design models.

8 Conclusion

In this paper, we studied an alternative way of structuring the design models of a MAS for an AGV transportation system using Theme/UML.

The results of our experience are promising. Separating the architectural concerns results in more coherent architectural description and allows to better investigate alternatives for particular concerns. We illustrated this by changing the distribution schema of the AGV transportation system. On the other hand, documenting different concerns separately makes it more difficult to have an overall view of the system.

Finally, the experiment shows that Theme/UML is not ready yet for architectural design. We propose several extensions to Theme/UML to support architectural design. Extensions include the integration of new types of design models and support for refinement. The proposed extensions are based on particular needs we experienced during the restructuring of the design models of the AGV transportation system. As future work, we plan to use Theme/UML for a number of additional architectural (re) designs. From these experiences we aim to extend Theme/UML for architectural design in a disciplined manner.

9 Acknowledgement

EMC^2 and Nelis Boucké are supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Thanks to Steven Op de beeck and Dimitri Van Landuyt for the discussions about this paper.

References

1. Weyns, D., Schelfthout, K., Holvoet, T., Lefever, T.: Decentralized control of E'GV transportation systems. In: International Conference on Autonomous Agents and Multi-Agent Systems, Industry Track. (2005) 25–29
2. Boucké, N., Weyns, D., Schelfthout, K., Holvoet, T.: Applying the atam to an architecture for decentralized control of a transportation system: Experience report. In: Quality of Software Architectures conference (QoSA). (2006)
3. Egemin, DistriNet: Emc^2: Egemin modular controls concept. (IWT-funded project with Distrinet and Egemin. http://emc2.egemin.com)
4. Egemin: Egemin website. (`www.egemin.com`)
5. Boucké, N., Holvoet, T., Lefever, T., Sempels, R., Schelfthout, K., Weyns, D., Wielemans, J.: Applying the Architecture Tradeoff Analysis Method (ATAM) to an industrial multi-agent system application. Technical Report CW431, Departement of Computer Sience, KULeuven (2005)
6. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. Addison-Wesley (2005)
7. Weyns, D., Helleboogh, A., Steegmans, E., De Wolf, T., Mertens, K., Boucké, N., Holvoet, T.: Agents are not part of the problem, agents can solve the problem. In: Proceedings of the OOPSLA 2004 Workshop on Agent-oriented Methodologies. (2004)
8. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice (Second Edition). Addison-Wesley (2003)
9. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures, Views and Beyond. Addison Wesley (2003)
10. Loughran, N., Rashid, A., Chitchyan, R., Leidenfrost, N., Fabry, J., Cacho, N., Garcia, A., Sanen, F., Truyen, E., Win, B.D., Joosen, W., Boucké, N., Holvoet, T., Jackson, A., Nedos, A., Hatton, N., Munnelly, J., Fritsch, S., Clarke, S., Amor, M., Fuentes, L., Pinto, M., Canal, C.: A domain analysis of key concerns known and new candidates. AOSD-Europe Deliverable D43, AOSD-Europe-KUL-6 (2006)
11. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley (2005)
12. Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software **January/February** (2006)
13. Early Aspects: Aspect-oriented requirements engineering and architecture design. (`www.early-aspects.net/`)
14. AOSD Wiki: Glossary. (`www.aosd.net/wiki/index.php?title=Glossary`)
15. Boucké, N., Holvoet, T.: Relating architectural views with architectural concerns. In: Accepted on Early Aspects at ICSE. (2006)
16. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: Proceedings of the International Conference on Software Engineering. (2004)
17. Harrison, W., Ossher, H.: Subject-oriented programming: a critique of pure objects. SIGPLAN Not. **28** (1993) 411–428
18. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-oriented design: towards improved alignment of requirements, design, and code. SIGPLAN Not. **34** (1999) 325–339
19. Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999) 107–119
20. Clarke, S.: Extending standard uml with model composition semantics. Science of Computer Programming **44** (2002) 71–100
21. Odell, J., Parunak, H.V.D., Bauer, B.: Extending uml for agents. In abd Yves Lesperance, G.W., Yu, E., eds.: Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence. (2000) 3–17

22. Clarke, S., Walker, R.J.: Composition patterns: an approach to designing reusable aspects. In: Proceedings of the International Conference on Software Engineering. (2001) 5–14
23. Smith, R.G.: The contract net protocol: high-level communication and control in a distributed problem solver. Distributed Artificial Intelligence (1988) 357–366
24. FIPA TC Communication: Fipa contract net interaction protocol specification (document sc00029). `http://www.fipa.org/specs/fipa00029/SC00029H.html` (2002) FIPA-Standard.
25. Schelfthout, K., Weyns, D., Holvoet, T.: Middleware for protocol-based coordination in dynamic networks. In: MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, New York, NY, USA, ACM Press (2005) 1–8
26. Schelfthout, K., Holvoet, T., Berbers, Y.: Views: Customizable abstractions for context-aware applications in manets. In: Proceedings of SELMAS'05, workshop at ICSE'05. (2005)
27. Atkinson, C., Kühne, T.: Stratified frameworks. International Journal of Computing Science and Informatics, Informatica **25** (2001) 393401
28. Atkinson, C., Kühne, T.: Aspect-oriented development with stratified frameworks. IEEE Software **20** (2003) 81–89
29. Garlan, D., Shaw, M.: An introduction to software architecture. In Ambriola, V., Tortora, G., eds.: Advances in Software Engineering and Knowledge Engineering, Singapore, World Scientific Publishing Company (1993) 1–39
30. Shehory, O.: Architectural properties of multiagent systems. Technical Report CMU-RI-TR-98-28, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA (1998)
31. Brussel, H.V., Wyns, J., Valckenaers, P., Bongaerts, L., Peeters, P.: Reference architecture for holonic manufacturing systems: Prosa. Computers in Industry **37** (1998)
32. Weyns, D., Holvoet, T.: Multiagent systems and software architecture. In: Special Track on Muliagent Systems and Software Architecture, Net.ObjectDays. (2006)
33. Kendall, E.A.: Role model designs and implementations with aspect-oriented programming. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (1999) 353–369
34. Kendall, E.A.: Role modelling for agent systems analysis, design and implementation. IEEE Concurrency **8** (2000) 34–41
35. Cabri, G., Leonardi, L., Zambonelli, F.: Modeling role-based interactions for agents. In: The Workshop on Agent-oriented methodologies at OOPSLA. (2002)
36. Garcia, A., Kulesza, U., Lucena, C.: Aspectizing multi-agent systems: From architecture to implementation. Software Engineering for Multi-Agent Systems III **LNCS 3390** (2004) 121–143
37. 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)
38. Op de beeck, S., Truyen, E., Boucké, N., Sanen, F., Bynens, M., Joosen, W.: A study of aspect-oriented design approaches. Report CW 435, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2006) URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW435.abs.html.
39. Chitchyan, R., Rashid, A., Sawyer, P., Bakker, J., Alarcon, M.P., Garcia, A., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of aspect-oriented analysis and design (2005) AOSD-Europe Deliverable No: AOSD-Europe-ULANC-9.
40. Katara, M., Katz, S.: Architectural views of aspects. In: Proceedings International conference on Aspect-oriented software development. (2003) 1–10