Relating Architectural Views with Architectural Concerns

Nelis Boucké* and Tom Holvoet
DistriNet, Department of Computer Sience, K.U.Leuven
Celestijnenlaan 200A, 3001, Leuven, Belgium
{nelis.boucke,tom.holvoet}@cs.kuleuven.be

ABSTRACT

Architectural views are at the foundation of software architecture and are used to describe the system from different perspectives. However, some architectural concerns crosscut the decomposition of the architecture in views. The drawbacks of crosscutting with respect to architectural views is similar to the drawbacks with respect to code, i.e. hampering reuse, maintenance and evolution of the architecture. This paper investigates the relations between architectural concerns, architectural drivers and views to identify why crosscutting manifests itself. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these concepts in an Online Auction system. Within this limited setting the first results look promising to better separate concerns that otherwise would crosscut the views.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Miscellaneous

General Terms

Design, Documentation

Keywords

architectural concerns, architectural drivers, views, slices

1. INTRODUCTION

Architectural design is generally considered as a crucial step to cope with the inherent difficulties of developing large-scale software systems. Software architecture is a key artifact since it embodies the gross-level structures and earliest design decisions that directly impact the expected quality attributes and subsequent design or implementation phases [8]. Aspect-Oriented Software Development (AOSD) enables modularization of crosscutting concerns [1]. Though originally related to the implementation stage, recently the support is extended to cover the whole development cycle (known as Early Aspects (EA) [2]). Dealing with concerns—having a high impact on the gross-level structures and quality attributes—must be done during architectural design, leading to an architecture-centric approach [9]. Such concerns are typically coarse grained and must be tackled on the architectural level, as recognized by the EA report [28].

Our position is that separating concerns —that otherwise would crosscut the decomposition— must also be applied on design models. Architectural views are at the foundation of architectural description and decompose this description in several design models. Each of these design models elucidates the system from a different viewpoints to emphasize certain facets. Unfortunately, some architectural concerns crosscut these views. The drawbacks of crosscutting with respect to architectural views is similar to the drawbacks with respect to code, i.e. hampering reuse, maintenance and evolution of the architecture. However, the relation between architectural views and separating concerns in AOSD remains largely unexplored. This paper starts with investigation of the relationships between concerns and views to identify why some concerns tend to crosscut. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these concepts in an Online Auction system. We believe that the main challenge for AOSD with respect to software architecture lies in advanced composition mechanisms, similar to extending UML with model composition semantics [12] or extending java with pointcuts as done by AspectJ.

Overview Section 2 introduces architecture, architectural drivers and the Online Auction system. Section 3 investigates the relation between concerns, drivers and views and identifies crosscutting. Section 4 proposes an extension. Section 5 discusses related work. Conclusions are drawn in Section 6.

2. ARCHITECTURAL CONCERNS

2.1 Software architecture

Software architectures covers the first design decisions to meet the essential quality requirements of the system. Often, the description is more abstract than UML class diagrams or sequence diagrams and will not necessarily have a one-to-one correspondence with code. 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 software components, 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 [14]. Each *architectural view* shows particular types of elements and the corresponding relationships between them. A *viewpoint* is a template from which to instantiate individual views [20]. Architectural views allow to separate concerns, since each view emphasizes certain facets of the solution as clear and concise as possible while deemphasizing and ignoring other facets. Many different viewpoints and sets of viewpoints exist, amongst them [24, 14, 18]. Views can be categorized according to three dimensions

*Supported by Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EA'06, May 21, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

(also called viewtypes) [14]: (1) views about the structure of implementation units (module); (2) views about the runtime units or runtime behavior and interaction (component and connector, C&C); and finally (3) views about how the software relates to its deployment and execution environment (allocation). No fixed set of views is appropriate for every system, but broad guidelines advise to include at least one view from each of these dimensions.

2.2 Architectural drivers

A *concern* is an area of interest or focus in a system. As stated in the AOSD Glossary [3], concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to a software engineer. From this, *architectural concerns* are defined as those concerns that significantly influence the architecture [9].

Because the set of architectural concerns is still rather ample and covers a lot, the architect will use a limit set (the headlines amongst the architectural concerns) to structure the architectural models and documentation. This set of high priority architectural concerns, called *architectural drivers*, will have a broad influence on the architecture, drive the architect while defining its architecture and make up large chunks of the architectural documentation. Architectural concerns are not considered to be a drivers if they have no broad influence, if they have no high priority or if they do not form a significant part of the architecture and architectural documentation (influencing several views and several architectural elements).

For example, in an Online Auction system the main architectural drivers could be the auction functionality (Auction), the security constrains that users must be logged on before they can take part in an auction (UserAccount) and the distributed nature of the system (Distribution). These architectural drivers are the 'headlines' amongst the architecture-related requirements[1] of the Online Auction system. Each of these drivers makes up a significant part of the architectural documentation and is described in more detail in the next section.

We avoid using the term architectural aspect [27]. It is with no doubt intuitive by exploiting the analogy with implementation-level concepts of Aspect Oriented Programming (AOP). Still, the analogy might turn out to be problematic [15], it could be assumed as straightforward translation of the concepts of the AOP model. Such direct translation is difficult since software architectures use completely different concepts.

2.3 Example: Online Auction system

An Online auction system is a client-server system to supports auctions on the internet. The users of the system are sellers and buyers Sellers can get a list of current auctions, create new auctions and cancel running auctions. Buyers can get a list of the current auctions, join or leave an auction and may place bids. The auction elapses as a simple Dutch Auction. Once the auction is started, buyers can place bids on the product. Each bid must be higher then the previous bid. At the end of the auction, the current best bid for the item is accepted. Both buyer and seller are notified that the auction has completed and the necessary bank transactions are forwarded to the bank (buyer is charged his last bid, seller is charged for the transaction costs). The system must be distributed (constraint) and be secure (quality attribute). The simple form of security tackled here is that users must be signed in before being able to participate in auctions. When creating an account, users must specify credit card information, username and password.

[1]Architecture-related requirements are those requirements with a significant impact on the architecture of the system [17]

From this requirements, several architectural drivers are identified. Firstly, the architect must identify the main modules and processes needed to list, create and run auctions (*Auctions driver*). This architectural driver involves the 'basic' functionality (auction procedure, interaction with users resulting from the auction) of the system and significantly influences the coarse grained structure of the program. Secondly, the architect must ensure that appropriate security measures are taken. Here, a simple mechanism to ensure that a user is signed in is used as an example (*UserAccounts driver*). UserAccounts influences the security guarantees of the system and has an influence on several other views of the system, which motivates considering it as driver. Finally, the architect must explicitly tackle the distribution of the system (Distribution driver), since it will have a far reaching effect on the structure of the architecture and on the existing views and architectural elements. For distribution, a client-server architectural pattern with a single server is used. Other examples of possible drivers, not further elaborated here, are Graphical User Interface (GUI) and Maintenance (the functionality provided by the software for the maintainer). How exactly the drivers are identified is considered outside the scope of this paper.
The Online Auction system is a relatively easy system, selected as example because the basic concepts of such type of system are well known and its (limited) complexity allows within limited space. Yet, such example provides first insights and paves the way to more complex examples.

3. RELATING DRIVERS AND VIEWS

Dealing with concerns must be done during architectural design, leading to an architecture-centric approach [9]. The key questions yet to be solved are how concerns and crosscutting relate to architectural views [2]. However, this relation is largely unexplored. An important observation for design is that not only crosscutting with respect to code can be problematic, but also crosscutting with respect to design models, in the case of architectural design the design models are architectural views. This section tries to characterize the relation between architectural drivers (as headlines amongst the concerns), architectural views and crosscutting with respect to architectural views. Section 3.1 elaborate on the problem of crosscutting, section 3.2 discusses the drawbacks of crosscutting with respect to architectural views.

3.1 Misalignment between drivers and views

Views are at the foundation of architectural documentation and are a mechanism introduced to handle complexity in the architecture by improving separation of concerns (SoC). However, there is a misalignment between the architectural drivers and the architectural views. Architectural drivers are the headlines amongst the concerns that drive the architect to describe the architecture. Architectural views are contain the architectural models and structure them according to several dimensions (as explained in the previous section. The problem is that architectural drivers tend to crosscut (are scattered and tangled over) the architectural views.

For scattering, two causes can be identified. Firstly, views are structured according to the module, C&C and allocation dimensions (as outlined in section 2.1) and a view typically uses a few types of elements and relations. Yet, architectural drivers do not necessarily align with the dimensions but will typically influence architectural elements in several architectural views and architectural dimensions. This leads to scattering of the elements contributing to an architectural driver over several views. For example, UserAccounts will manifest itself in several dimension by defining both the structure (module dimension) and behavior (C&C dimen-

sion). Secondly, elements of a single concern can be scattered over several views (even if they are from the same dimension) as a consequence of inherent complexity of large-scale systems. For example, if process view (an example of a process view can be found in Figure 1) contains to many processes the whole system becomes cluttered and the natural result is to make several views describing coherent subsets of the processes.

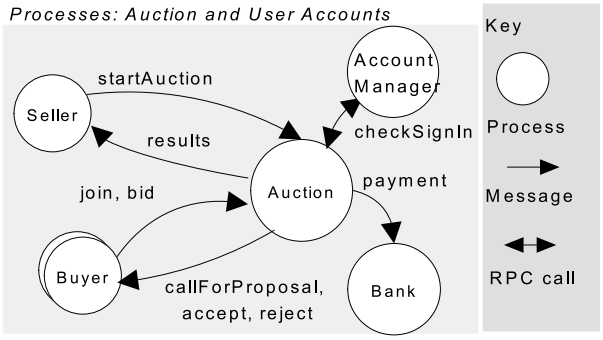

Figure 1: Process views illustrating background processes of the system.

Tangling occurs in architectural views if a single view contains parts of descriptions of several architectural concerns. Which elements and relations to record in a single view is the software architects decision. Also, an architectural representation is rather flexible since it allows to create hierarchical designs (part of the design at a higher level is refined in a lower level). At first glance this seems to imply that tangling of concerns in views can be easily prevented by limiting architectural elements in a view to a single architectural concern. A closer look reveals that much remains to be done before architectural views are really untangled.

There is an incentive to structure the description according to the dimensions of views. However, there is no incentive to structure the elements in the views to better align with the architectural drivers (since drivers are not made explicit and not directly related with the views describing them), leading to bad traceability.

Even with such incentive, there remain problems to be tackled. The problem lays mainly in the relationships between views or how the system can be composed from the separated views. One problem appears when a connector is needed between to elements of two separate architectural drivers (connector problem). The only way to connect the elements is by including them in the same view, i.e. in one of the original views or a in new view view containing the elements and the connector between them. This obviously leads to tangling. For example, Figure 1 contains the process of both Auction and UserAccounts that can not be described without tangling because all processes are linked together with connectors[2]. A second problem is that relations between architectural views are often implicit, e.g. elements having the same name in different views are often assumed to be the same element. Other relations are informal, e.g. textual and informal descriptions of related view packets. Additionally, the architectural documentation can contain information beyond views [14], including a mapping between elements of several views and an element directory recording which element appears in which view. The type of relations described until now suffice when considering classical use of architectural views. But when deepening the separation between views to better align with the architectural drivers, more powerful mechanisms are needed to describe the relationships. Additionally, we would like more explicit and formal relations between views so that tools can be used for on demand composition of models. Without such mechanisms, software architects are not encouraged and are not able to make untangled views.

[2]Graphical elements of figures are only defined once in some key. Also consult keys of previous figures and the text explaining the figure.

3.2 Drawbacks of crosscutting

The drawbacks of crosscutting with respect to the architectural descriptions are similar to the drawbacks on the level of code. Firstly, since no single architectural view or set of views is identifiable as description of the architectural driver, advantages of distinct design and development are lost. Before being able to update the design for a particular architectural driver, an architect must review all views because there are few guidelines where to search. This clearly prevents traceability from architectural drivers to architectural elements and hampers maintenance and evolution. Secondly, the standard notion of views does not allow explicit definitions of 'open spots' (like abstract classes or parameters) that should be filled in later. Together with the traceability problem this hampers reuse of architectural designs in other applications.

4. EXTENDING THE ARCHITECTURAL DESCRIPTION

Starting from the current notion of views and the problems identified in the previous section, we propose to extend the architectural description with slices and composition mechanisms. The proposal is initial in the sense that the concepts are only initially explored in a limited setting.

4.1 Extensions

Our first proposition is to extend the architectural description with the concept of an *architectural slice* (similar to hyperslices in HyperJ for implementation [26] or Themes for detailed design [6]). There are two types of slices: primitive and compound slices. *Primitive slices* are single views. *Compound slices* group several other slices (primitive or compound) together to cover a specific architectural driver. The advantages of architectural slices are twofold. Firstly, since the architectural elements in a slice are meant to cover a specific driver, there is a direct traceability between drivers and the views describing them (and thus no tangling). For compound slices, the still exists scattering over several subslices, but they are clearly grouped together and directly traceable from the drivers. Secondly, hierarchical composition allows to gradually buildup the design of a large scale system. Such hierarchical composition can be very beneficial for the scalability of the approach.

Our second proposition is to allow architectural slices to have 'open spots' under the form of abstract elements (like UML or in [19]) or *parameters* (like in Theme/UML). These parameters can be bound to concrete values of other slices when describing the composition. Note that a single parameter can be bound to several concrete values, which is central to AOSD, i.e. to apply one concept at different places simultaneously. Introducing such 'open spots' is needed to cope with the reusability issues outlined in the previous section. Some of these parameterized slices could be seen as architectural patterns [10]. Parameterization will also help to tackle the connector problem of section 3.1 by defining a connector on a parameterized element and later on bind this element to the concrete element of another slice.

Finally, our third proposition is to make composition and relations more explicit in the *slice composition diagram*. In this diagram, slices are handled as first class elements and a connector

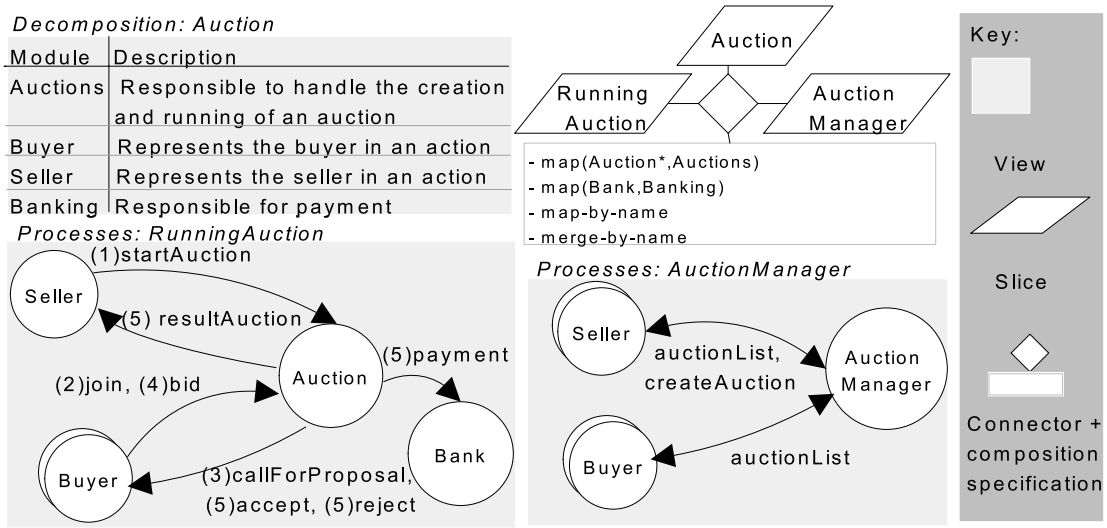

Figure 2: The Auction slice.

encapsulates the composition specification. Extending the architectural description with explicit slice connectors is needed to allow the two previous extensions (powerful composition mechanism and parameterization) and to make relations more explicit.

4.2 Illustration in Online Auction system

The concepts of slice, parameterization and composition specification are illustrated using a partial architectural description of the Online Auction system. Each architectural driver are tackled by a separate architectural slice.

4.2.1 Slices and parameterization

Consider Figure 2 describing the Auction slice, containing three primitive slices (views) and a composition diagram. The Auction slice could be considered as the 'base' slice because it covers the primary functionality of Online Auction system. The slice describes listing and creations of auctions, the auction procedure and interaction. The subslices are: (1) *Auction*, a module decomposition view identifying the static modules for the auction and their responsibilities. (2) *RunningAuction*, a process view describing the auction procedure and interaction between those processes involved in a running auction. The numbers before the calls or messages indicate the sequence followed (these numbers could be omitted). If several calls or messages have the same sequence number, the order between them does not matter. (3) *AuctionManager*, a process view describing the process involved when creating and listing auctions. The composition diagram is explained further in this section.

The UserAccounts slice contains three views and is described in Figure 3. Especially the view *CheckSignIn* is of interest because it introduces two new concepts: parameterization and interaction refinement. Parameterization is indicated by underlining the names of processes, calls or messages. The forked arrow indicates interaction refinement (inspired by [5], see related work). Above the arrow is the description of the original interaction, below the arrow is the refinement of this interaction. The part above the arrow (a situation description) together the binding rules roughly corresponds to a pointcut. The part below the arrow (the refined interaction) can roughly be seen as the advice. Thus, the *CheckSignIn* view describes that before allowing any message from the user process to process X, process X will check that the user is properly logged in.

4.2.2 Slice composition diagram

A slice composition diagram contains a description on how the slices must be combined. Slices are represented by parallelograms, relations between slices by lines leading to a diamond. Each relation is annotated with additional composition information in the form of composition rules. The composition rules specify how the slices should be connected with each other. When reifying the composition, rules are applied from top to bottom.

During the composition specifications of the Online Auction system several composition rules have been used. But first some syntax: `expr` indicates an expression build up with text and wildcards (indicated by an asterisk). Brackets indicate that several expressions are listed. E.g. 'auction*' will match elements starting or ending with 'auction', '{auction*, *auction}' will match both elements starting or ending with 'auction'. `element` indicates that only a single element can be filled in.

- `match(expr,element)` : used to resolve naming differences between elements in different slices. Matching elements is only possible between elements of the same type (e.g. match process with process or module with module). An example can be found in Figure 4 where Seller and Buyer are matched with User[3]
- `map(expr,element)` : maps (multiple) element(s) on a single element. This composition rule can describe the same type of relations as the 'Mapping between views' tables of [14]. For example in Figure 2 this rule is used to map the Auction and AuctionManager process (who both match Auction*) onto the Auctions module. `map-by-name` will do a standard mapping between the elements based on names. As second example, consider Figure 5 where modules are mapped on other modules (thus forming sub modules).
- `generalization(expr,element)` : generalization relations between the elements who match the expression and the element in the right hand side of the relation. In our example, the composition rule is used to specify that user is a generalization of seller and buyer. In Figure 4, a visual notation is used to describe that user is a generalization of seller and buyer.

[3]The generalization rules does not imply anything about the processes, thus the match expression is not redundant. Buyer and Seller can be submodules from User without implying that Seller, Buyer and User are the same process.

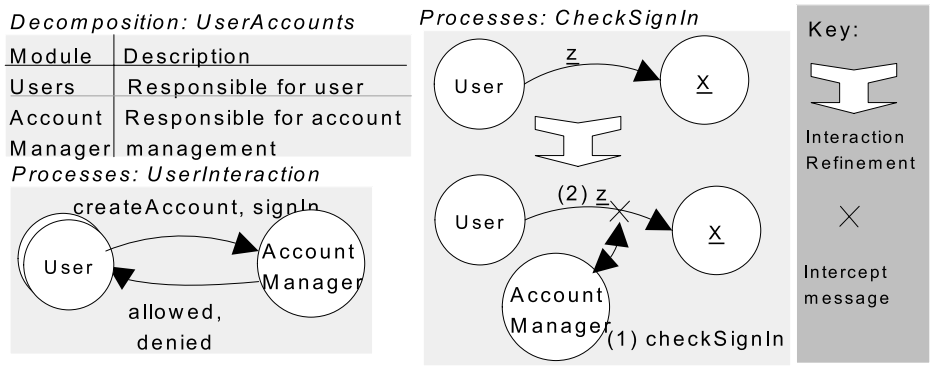

Figure 3: The UserAccount slice.

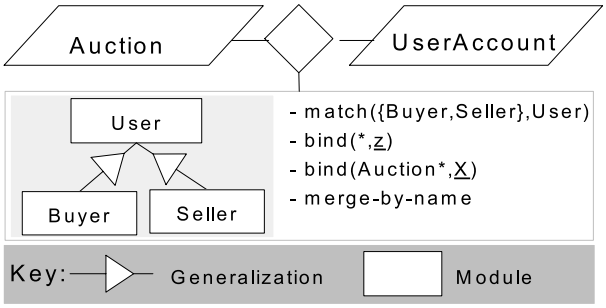

Figure 4: The AuctionAccount slice, merging the slices Auction and UserAccount together.

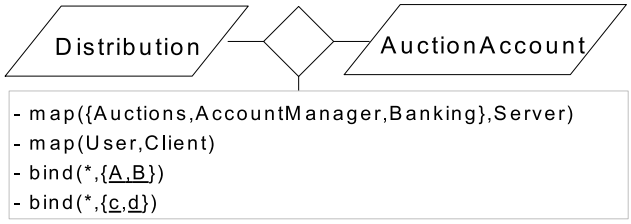

Figure 5: Composition of AuctionAccount slice and Distribution slice.

- `bind(expr,parameter)` : binds elements to the specific parameter. For example, in Figure 4 this relation is used to bind 'Auction' and 'AuctionManager' to X. Process X is a process for which the user must be logged in before it can be accessed.
- `merge(expr,expr)` : merges elements of the first expression with elements of the second expression if they are from the same type, similar to the merge operation defined for Theme in [12]. `merge-by-name` will merge elements on their names.

If no composition rules are specified, `map-by-name` together with `merge-by-name` are assumed. `match-by-name` is always automatically used, manual matching will only add additional matches. If some parameters are left unbound, they are just taken to be parameters of the the new composed slice (not used here). No difference is made between the reconciliation (resolve compatibility problems between different slices) and composition of the concerns, reconciliation can easily be defined as composition rules (e.g. match).

4.2.3 *Distribution influences both Auction and UserAccounts*

As an additional example, consider Figure 6. We elaborate on distribution because it provides a good example of a compound slice (distribution) that influences two other compound slices (Auction and UserAccounts, merged in the AuctionAccount slice).

The Distribution slice has four subviews. The *Subsystems* view decomposes the system in a client, server and communication subsystems. Both the client and server can use the communication subsystem, described in the usage view *Communication*. The *ClientServer* view shows how client, server and communication are deployed on several computers connected by a TCP/IP network.

Finally, the combined view *RemoteMessages* shows that the connector between process A and process B (sending messages) must be refined and that messages must be sent through the communication subsystem. This communication subsystem ensures that the messages are sent through the TCP/IP network (e.g. taking care of appropriate message format, appropriate host addresses, compression, etc.). Putting the process in the module in a combined view is equivalent to mapping processes on modules.

Figure 5 shows how the distribution slice is combined with the AuctionAccount slice. Care should be taken when interpreting the composition specification. The two first rules do map the appropriate modules onto client and server (forming submodules). This also implies that all processes that previously have been mapped on these submodules, now also belong to this module. Thus the processes of client are Seller and Buyer (and the generalization in User); the processes of server are Auction, AuctionManager, Bank and AccountManager. The next two rules might suggest to map any process on A and B and any message on c and d, but the constraint that A belongs to the Client subsystem and B belongs to the Server subsystem from the *RemoteMessages* view should also be taken into account. Constraints imposed by the view are applied before the composition rules. Thus Seller and Buyer (and their generalization in User) are bound to A and Auction, AuctionManager, Bank and AccountManager are bound to B. Bindings for c and d follow a similar approach.

4.3 Discussion

Quantification, an essential property of AOSD [16], over architectural elements is found in the expressions used in the composition rules. In the example, the expressions are rather limited (text and wildcards), but they can easily be extended. An important observation during the exploration is that rules tend to have different influences on different types of elements. For example, the mapping rule can be used to map process on modules, modules

on computer systems and modules on modules (defining them as being submodules). Notice interaction refinement is currently used *within* the slice and not in the composition specification, although we think it is possible. We decided to include it in the view because this visual notation is very intuitive. Also, only a limited set of rules have been applied. Interesting possibilities to consider in the future are override composition operator (used in Theme/UML) and considering interaction refinement (used in Architectural Stratification, see related work) as a composition rule. A final remark on composition rules is that they strongly depend on the constraints imposed by the views (as illustrated with the composition rules of Figure 5) and that they may interfere with each other. Currently, rules are only informally defined and it is clear that further research is needed to fully understand the possibilities and implications of using such rules to compose slices together. E.g. apply richer sets of composition rules; studying richer expressions in the rule parameters; formalizing the composition rules, their implications on the architectural elements and the process of applying the rules on the description language.

One of the main innovations is to make the composition diagrams part of the architectural description. Each composition diagram specifies a new compound slice, which could be reified by employing the composition specification. The architect could decide to reify a compound view to understand it better, or to incorporate it in the documentation for clarity reasons. Appropriate tool support can make such reification easier.

Until now, a single connector per connection diagram is used. For example, Auctions and UserAccounts are composed in a slice called AuctionsAccounts. Afterwards this compound slice is composed with the Distribution slice. An alternative is is to compose the system with to connectors in a single slice composition diagram. Notice that the scope of a composition rule is limited to the slices linked with the connector in which the rule is specified. Composing all three slices in one diagram would keep the both connectors and composition rules of Figure 4 and Figure 5, the only change is that the latter must be reconnected to Auction and UserAccount instead of AuctionAccount. Further research is needed to understand the full implications of using multiple connectors in a composition diagram.

Mapping to previous and following development phases is not explicitly considered in this paper. Investigating the exact relations with concerns identified in requirements and concerns tackled during detailed design remains to be done.

5. RELATED WORK

A good survey on design approaches (both architectural and detailed design) can be found in [11]. Here, we only focus on approaches that are closely related to our work.

Rozanski and Woods [25, 29] identify that quality properties (for example security) appear in several architectural views. This is clearly related with the identification of crosscutting in the previous section. The authors introduce architectural perspectives as complementary to architectural views in the sense that they define a set of activities, tactics and guidelines to ensure that the system exhibits a particular quality property. Architectural perspectives is not a technique for modular description but rather a framework to guide and formalize the process of ensuring that a particular architectural property is met, perspectives are *applied onto* views.

Atkinson and Kühne [5] propose architectural stratification to combine the strengths of component-based frameworks and model-driven architectures (MDA [4]) to support AOSD. The approach starts from the observation that you can define several structural architectures - depending on the level of abstraction at which you would like to see the system - each with different sets of connectors and components. The goal is to identify, elaborate and relate the architecture on these different levels. The architectural description is structured according to architectural strata, which is a full description of the system on a specific level of abstraction (strongly related to architectural views). Architectural strata represent different levels of architectural decomposition and lower strata refine the connectors of higher strata. The authors suggest that every stratum can be used to tackle a specific concern. Current limitations of the approach are that it supports only one type of relation (interaction refinement), the rigid structure of architectural strata allowing only relations with the stratum above and below and limiting the description on a stratum to a single view. Our proposal, on the contrary, supports multiple types of relations, supports multiple views and has a less rigid structure.

In [22] Kande et al. study the need for multidimensional SoC in architecture descriptions. [21] proposes a concern-oriented framework called Perspectival Concern-Spaces (PCS). The goal is to develop architecture with as primary dimension the concerns, using an extension to UML as modeling language. But the PCS uses a very specific interpretation of IEEE-Std-1471 [20] (in which viewpoints are concern spaces) that differs substantially from the generally accepted interpretation of viewpoints in software architecture (e.g. described in [18, 25]). The interesting point raised by this work is the relation between MDSOC [26] and architectural views.

Theme [6, 12, 13] is an analysis and design approach proposed to cope with the structural mismatch between requirements and design using UML. For aspect oriented design, the approach defines a UML based aspect oriented design language called Theme/UML, extending the UML meta-model with model composition semantics. Concerns, crosscutting or not, can be separated in Themes (forming a symmetric approach). Every Theme contains several partial UML-models which are closely related to each other and can be parameterized. The most important contribution of Theme/UML is the explicit composition of Themes (UML-models) with composition operators like bind (binding concrete constructs to the parameters), merge and override. The differences with our approach is that Theme acts on detailed design level, e.g. on class diagrams and interaction diagrams and relying on all details of them. Secondly, Theme uses an explicit reconciliation phase while for our approach reconciliation is part of the composition rules. Finally, Theme uses only one composition rule to combine Themes while our approach allows specifying multiple rules.

Katara and Katz [23] 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 authors propose an extension to UML and a new architectural viewspoint (called concern diagram) describing how aspects can be combined to tread different concerns of a system. The concern contains dependency relations between aspects and shows which aspects contribute to which concerns. The work is related to our approach in the sense that relations between several aspects and concerns are made explicitly on the design level. Where Katara et al. mainly focusses on overlapping and interference between aspect in UML models, we focus on adding several types of non-trivial relations in the architectural description.

A recent paper of Baniassad et al. [7] also considers views as primary decomposition on the architectural level and identifies concerns crosscutting several views. The authors introduce the concept of an aspect view to capture concerns that otherwise would crosscut the decomposition in views. The difference with classical views is that aspects can contain abstract elements which is similar to the parameterization used in our approach. The difference is in

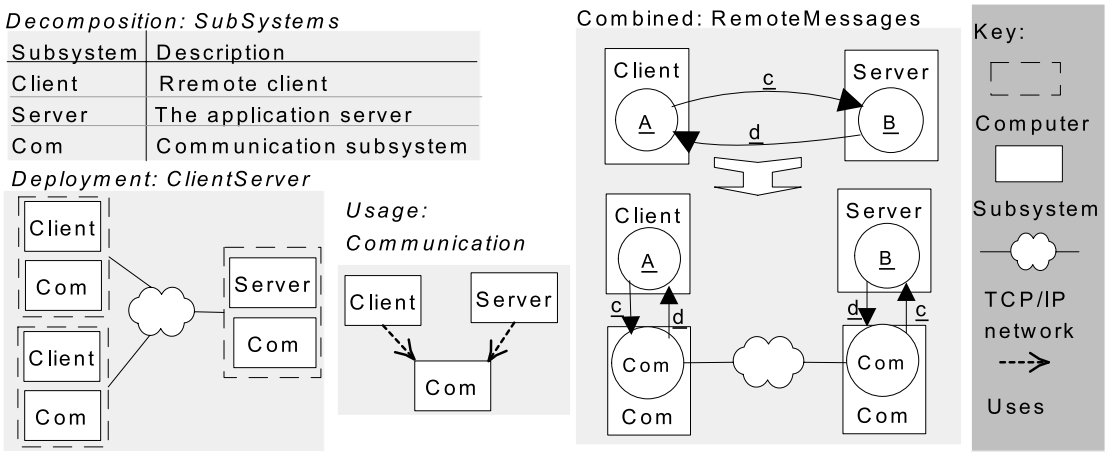

Figure 6: The Distribution slice.

the concept of a slice as hierarchical composeable design building block and the explicit composition diagrams with connectors and composition rules used in our approach.

6. CONCLUSION

This paper investigates the relations between architectural concerns, architectural drivers and views and identified that concerns tend to crosscut due to structural mismatch between concerns and views and limited support for composition between views. We propose to extend the architectural description with slices and composition mechanisms to prevent this crosscutting and perform an initial exploration of these concepts in an Online Auction system.

Within this limited setting the first results look promising to better separate architectural drivers that otherwise would crosscut the views and to prevent the typical drawbacks of crosscutting. One of the first steps for future research is to more exactly define the composition process, the composition rules and their implications on architectural elements. Especially the composition rules ask for further research, e.g. apply richer sets of composition rules and study richer expressions. Additionally the implications of applying the extended architectural description to more complex case studies is an important challenge for future research.

7. REFERENCES

[1] Aspect Oriented Software Development (AOSD) website. `www.aosd.net`.

[2] Early aspects: Aspect-oriented requirements engineering and architecture design. `www.early-aspects.net/`.

[3] Glossary from AOSD wiki. `www.aosd.net/wiki/index.php?title=Glossary`.

[4] Model driven architecture (mda) website. `www.omg.org/mda/`.

[5] C. Atkinson and T. Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.

[6] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, 2004.

[7] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Software*, January/February, 2006.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architectures in Practice (Second Edition)*. Addison-Wesley, 2003.

[9] N. Boucké and T. Holvoet. Dealing with concerns ask for an architecture-centric approach. In *European Interactive Workshop*, 2005.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[11] R. Chitchyan, A. Rashid, P. Sawyer, J. Bakker, M. P. Alarcon, A. Garcia, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design, 2005. AOSD-Europe Deliverable No: AOSD-Europe-ULANC-9.

[12] S. Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44(1):71–100, 2002.

[13] S. Clarke and R. J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the International Conference on Software Engineering*, pages 5–14, 2001.

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

[15] C. E. Cuesta, M. del Pilar Romay, P. de la Fuente, and M. Barrio-Solorzano. Architectural aspects of architectural aspects. In *2nd European Workshop on Software Architecture (EWSA)*, volume LNCS 3527, pages 247–262, 2005.

[16] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[17] D. G. Firesmith and P. Capell. Architecture-related requirements. *Journal of Object Technology,*, 5(2):61–73, March-April 2006.

[18] J. Garland and R. Anthony. *Large-Scale Software Architecture, A practical guide using UML*. Wiley, 2003.

[19] S. Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML*, 2002.

[20] IEEE. Recommended practice for architectural description of software-intensive systems (ansi/ieee-std-1471), September 2000.

[21] M. Kandé. *A Concern-Oriented Approach to Software Architecture*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.

[22] M. M. Kande and A. Stroheier. On the role of multi-dimensional separation of concerns in software architecture. In *Prooceedings OOPSLA workshop on Advanced Separation of Concerns*, 2000.

[23] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings International conference on Aspect-oriented software development*, pages 1–10, 2003.

[24] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[25] N. Rozanski and E. Woods. *Software Systems Architecture*. Addison Wesley, 2005.

[26] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Int. Conf. on Software Engineering*, pages 107–119, 1999.

[27] B. Tekinerdogan. ASAAM: Aspectual software architecture analysis method. In *Early Aspects*, 2002.

[28] B. Tekinerdogan, A. Moreira, J. Araujo, and P. Clements. Workshop report of Early Aspects at AOSD, 2004.

[29] E. Woods and N. Rozanski. Using architectural perspectives. In *Proceedings of the WICSA conference*, 2005.