A Generalization of the Hyperspace Approach Using Meta-Models

Daniel Lohmann, Jürgen Ebert
University of Koblenz-Landau
Institute for Software Technology
Universitätsstraße 1, D-56070 Koblenz
{daniel, ebert}@uni-koblenz.de

Abstract

In this paper we describe a generalization of the Hyperspace approach of TARR and OSSHER that is applicable to design description languages. We understand artifact languages described by meta-models as the primary dimensions of a multi-dimensional concern space. This space is filled by concrete concerns and by concrete artifacts to build an integrated graph-based multi-paradigm design description. Secondary dimensions may be added to describe non-artifact-based concerns. User-defined views to and extraction of concern subsets is facilitated by a transitive closure approach.

1. Introduction

During software development, a lot of different concerns have to be taken into account. New concerns come in from very different sources and show up in every stage of the development process [1]. Depending on persons and stages these concerns are expressed in different ways. For example, one typical form of concern in object-oriented design is expressed by *classes*; during analysis we usually speak about concerns like *features* or *performance requirements*, etc.

These different kinds of concerns can be understood as different *views* to the system. TARR and OSSHER introduced the idea to interpret these views as *dimensions of a concern space* and to do *multi-dimensional separation of concerns (MDSoC)* on this integrated model [2, 3]. They call this approach the Hyperspace approach.

Concerns are usually described in some form of formalism. We call these formalisms *artifact languages*. Like a dimension in the Hyperspace, an artifact language describes a specific view on the software system. For example, UML provides a set of sublanguages that are used for different views (e.g. static view, dynamic view …) at different stages.

We are convinced that there is a strong relationship between views/dimensions on the one side and artifact languages on the other side. For successful MDSoC over the whole software development cycle, one therefore needs a model that supports on-demand integration of artifacts and artifact languages. However, current attempts for MDSoC and aspect-oriented software development (AOSD) mainly focus on single languages and mainly on the implementation phase. Our goal was to find a model that supports MDSoC by language integration already in the earlier stages of software development, especially dealing with the languages of software design.

In this paper we introduce an approach for MDSoC by language integration using meta-models. Our approach is based on the Hyperspaces idea of OSSHER and TARR, which also claims to support MDSoC over all stages of the software development cycle [3, 4]. However, the current instantiation in Hyper/J again supports only one type of artifacts (namely Java .class files) and thus it emphasizes the implementation phase. Our investigations showed that some constraints of the formal Hyperspace model are too strict for less formal languages like the ones typically used in software design. Hence we decided to extend and generalize the Hyperspaces idea with a focus on language integration of design languages.

This paper is organized as follows: First, we introduce the terminology and basic concepts of the Extended Hyperspace model in section 2. In section 3, the model is then put into practice by an example. Section 4 discusses the main differences to the original Hyperspace model and gives an overview of other related work. Finally, in section 5 we conclude our work.

2. The Extended Hyperspace Model

In this section we introduce the Extended Hyperspace model. As mentioned above, our model is based on the Hyperspace idea by OSSHER and TARR. We use the same basic concepts and terminology where appropriate.

2.1. Basic Concepts and Terminology

Software consists of *artifacts*, i.e documents describing the concerns of a software system. Artifacts are written in *artifact languages,* like the UML family of languages and its sublanguages (e.g. class diagrams, use-case diagrams …).

An ideal artifact language describes just one kind (dimension) of concerns. While this is (mostly) true for the UML languages, other real languages may be a combination of more than one ideal language. In this paper we assume that we have to deal with ideal languages only.

Each artifact language consists of a set of syntactical constructs that can be used to build concrete artifacts. We call these syntactical constructs *unit types*. Examples for unit types in UML class diagrams are *class, attribute* and *method*.

Syntax and semantics of an artifact language itself has to be defined in some formalism. For the visual languages used in software design, *meta-modeling* is a suitable and common way to do so[1]. Here we use meta-modeling for describing the *abstract syntax* of artifact languages. Each syntactical concept (unit type) of the language is represented by a meta-class in the meta-model, and each relationship between them is represented by a meta-association. However, the abstract syntax makes no assumptions about shaping and layout of units and relationships.

[1] For text-oriented artifact languages, one would probably use formal grammars for similar purposes.

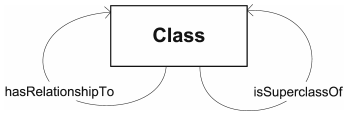

Figure 1 A meta-model of (UML) class diagrams

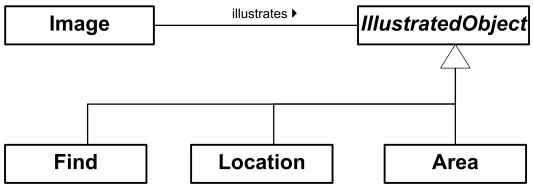

Figure 2 A class diagram artifact

For illustration purposes, we use very simple and minimal meta-models here. These meta-models are intentionally kept simple for comprehensibility reasons; the whole approach should work for real-life meta-models as well. Our meta-models contain only those unit and relationship types that are actually used in the concrete artifacts of the example. We also omit all additional properties of the unit and relationship types like cardinalities, constraints etc. Figure 1 shows the meta-model for class diagrams. It consists of only one unit type *class* and two relationship types: *isSuperclassOf* for super-/subclass relationships and *hasRelationshipTo* for associations. This is a fairly simple meta-model, but powerful enough to instantiate artifacts like the class diagram shown in Figure 2.

Similarly, a meta-model for use-case diagrams is defined in Figure 3 with a concrete use case diagram being shown in Figure 4.

2.2. Integrating Dimensions into a Concern Space

In the (Extended) Hyperspace model, units and concerns are organized in a concern space. A concern space consists of a set of dimensions, a set of concerns and a set of units. Each *concern* is placed in exactly one *dimension*. Each *unit* is mapped to zero or more concerns, namely the concerns it addresses.

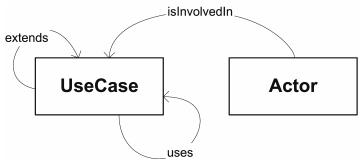

Figure 3 A meta-model of use-case diagrams

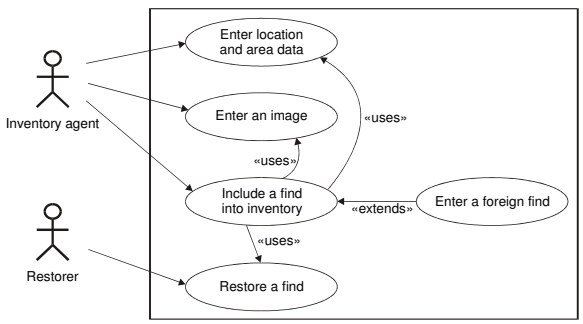

Figure 4 A use-case diagram artifact

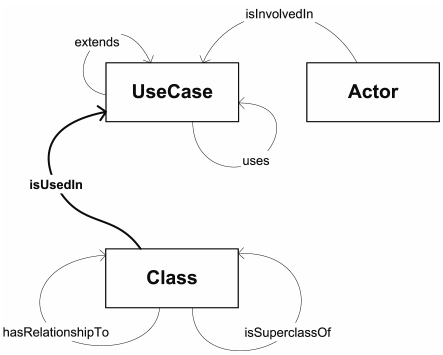

Figure 5 An integrated meta-model for class and use-case diagrams

The aim of a *concern space* is to integrate units and concerns of a software system in such a way, that concerns can be easily identified and separated, that relationships between different concerns become clear and that a software system can be built out of selected concerns [3].

As stated above, we understand artifact languages as dimensions of a concern space. Thus, a concern space can be seen as an integrated and multi-paradigm design description of a software system. It is built by integrating all the dimensions of concerns (*artifact languages*), and integrating the descriptions of concrete concerns (*artifacts*) into it.

Integration of artifact languages corresponds to integration of the underlying meta-models [5]. Figure 5 shows this integration for our meta-models of class and use-case diagrams. Here, the integration is done by adding an *additional relationship* that connects unit types from different languages. A new meta-association *isUsedIn* has evolved. Its intention is to connect a class to those use-cases that utilize services of the class e.g. by working on its instances.[2] This integration information is the key benefit of integration; it provides extra knowledge about inter-dimensional connections and dependencies that is not available from the original artifacts. We will use it later on for selecting and separating concerns.

2.3. Secondary Dimensions of Concerns

The dimensions we have taken into account so far are all based on artifact languages. Each artifact language used to describe concerns of our software system opens up one dimension of concerns. For example, the concerns modeled by class diagrams are classes, so we place each class as a concern on a <Classes> dimension and map all units to it that describe it: Methods units, attribute units, and, of course, the class unit that represents the concern itself. We call these artifact-based concerns *primary concerns* and their dimensions *primary dimensions*, respectively.

Note that we are distinguishing between *class unit* and *class concern.* The first one is a syntactical construct provided by the artifact language; the second one is an intentional construct of our mind. However, often there is a one-to-one relationship between them: Each class concern usually coincides with exactly one class unit, i.e. the unit is a *syntactical representative* of the

[2] A formal background for meta-modeling of abstract syntax including the meaning of meta-model integration is given in [5,11].

modeled concern. Nevertheless, real world software models generally also contain class units that do not represent concerns but merely are (language dependent) technical helper constructs for implementation details.[3]

An ideal artifact language provides exactly one representative unit type. Since they are representing primary concerns, these units are also designated as *primary units*.

Analogously, the concerns modeled with use-case diagrams, namely use-cases, can be placed on a <UseCases> dimension. Here, we again have to distinguish between *useCase units* and *useCase concerns*.

There will always be concerns and dimensions which are not (yet) represented by their own artifacts and artifact languages. Special concerns stemming from the application domain or "on-demand" dimensions for additional user- or task-oriented views are examples for such concerns without underlying artifacts. In our model we put such *secondary concerns* on their own *secondary dimensions.*

Being not based on artifact languages, secondary dimensions are not meta-modeled. Consequently, they are neither represented in the integrated meta-model, nor does there exist any describing artifact that supplies (primary) units for them. However, we can always map the units already introduced via primary dimensions to the new secondary concerns they relate to. Doing so, we get additional views to the concern space. Thus, secondary dimensions provide *on-demand* and *alternative separation of concerns* along arbitrary dimensions.

3. Example: An Inventory System

After introducing basic concepts and terminology, we now show how to use the extended model by an example. We integrate two different design artifacts, define secondary dimensions for alternative views and use the integrated model to generate concern-specific slices of the whole system.

The scenario is an inventory system for the State Office of Cultural Heritage. The mission of its archeological department is to salvage, restore, and inventory remains of former human cultures like potsherds, bones or tools. Such objects of archeological relevance, the *finds*, are often discovered during building and excavation works. The archeological department maintains a *find archive*, for which an inventory system shall be developed.

3.1. The Artifacts

The inventory system is described by two different artifacts: A class diagram and a use-case diagram.

Figure 2 shows a small part of the class diagram. *Find*, *Location* and *Area* are the main objects of interest that are stored in the inventory system. Each of them can optionally be illustrated by *Images* (e.g. photos or drawings). As usual, common properties like that are extracted into an abstract base class, which is *IllustratedObject* here.

Figure 4 shows a part of the use-case diagram for the inventory system. The main use-case is *Include a find into inventory*. It describes the process of taking a find (an object that has been found somewhere), entering its properties (material, state, estimated age, etc.) into the system and finally storing it in the find archive. During this process it may also be necessary to restore or conserve a damaged or decaying object (*Restore a find*). Additionally, if not already present in the system, the location and area data where this object has been found (*Enter location and area data*) and related images (*Enter an image*) are entered.

Enter a foreign find describes a variant of *Include a find into inventory* where the same data is recorded. However, it is not stored in the archive, because the object itself is owned by somebody else or even not present at all.

3.2. Artifact Integration

The artifacts describe two single dimensions of concerns, namely <Use-cases> and <Classes>. We integrate them as primary dimensions into a concern space. We do so by determining the described concerns and then assigning each unit to the concerns it addresses. The integrated concern space is shown in Figure 6. It contains two primary dimensions <Use-cases> and <Classes> that are depicted as clusters of concerns and their related primary units. Additional units are assigned by solid arrows. (For now, please ignore the different levels of grey and the secondary dimensions <Features> and <Tasks>.)

Every use-case of our use-case diagram models a concern on its own, but not every class from the class diagram. This is because the class *IllustratedObject* is not a *concern* of our application. It is merely a *technical construct* to implement polymorphic behavior and generalization.

As mentioned above, the integration of primary dimensions corresponds to the integration of the underlying artifact language meta-models. The resulting integrated meta-model has already been shown in Figure 5, it introduced a new meta-association *isUsedIn* that connects class units to those use-case units which utilize services of the class. Now, during artifact integration, we have to instantiate this meta-association to relate concrete class units to concrete use-case units. In Figure 6 this is illustrated by dashed arrows between units.[4]

3.3. Defining Secondary Dimensions

Secondary dimensions provide additional concerns that are not based on existing artifacts. We use them here for a more user-centric view to our software system. The concerns of typical end users are usually not expressed by class- and use-case diagrams; users do rather look at a software system in functional terms like *Features* and *Tasks*.[5]

A *feature* is a concern of a software system that stems from the functional domain of the end user. System specifications often

[3] Application of a design pattern like Observer is a good example. In Java this is typically realized by inheritance, needing additional interfaces and helper classes.

[4] Note that Figure 6 shows only those units that are assigned to concerns or take part in new relationships like *isUsedIn* which are not already present in the original artifacts. This is for legibility purposes. However, the omitted units and relationships (e.g. the class *IllustratedObject* and its inheritance relationships to *Find*, *Area* and *Location*) are considered to be still present in the concern space.

[5] Of course, organizing concerns on a secondary dimension is in general only second best compared to describing them by real artifacts. However, they are useful if you have not found an adequate formalism to express these concerns yet. Therefore, secondary dimensions can be seen as an indicator for missing artifacts or even missing artifact languages.

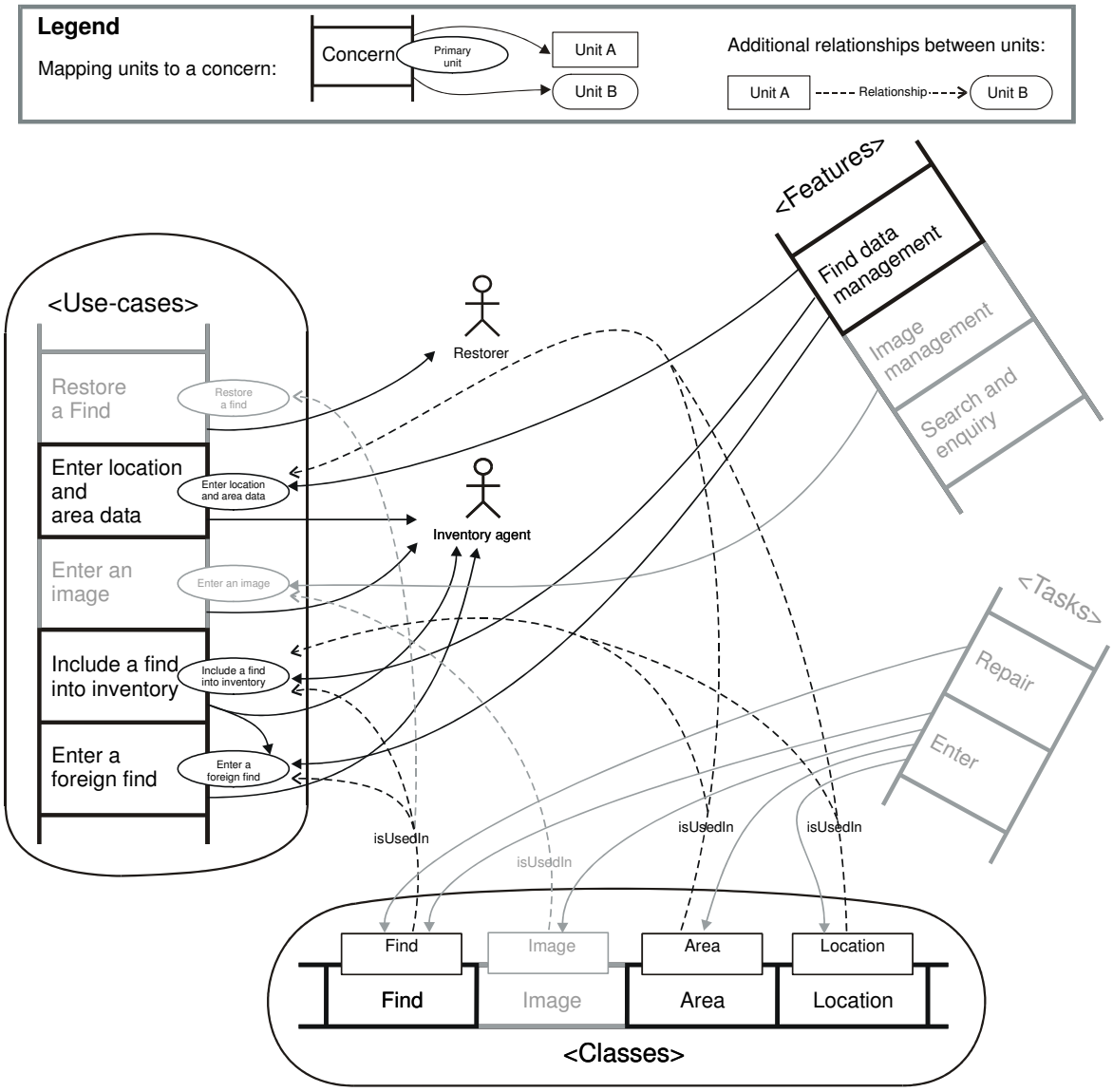

Figure 6 Concern space of the inventory system

consist of feature lists; release cycles are driven by features to implement. Features are used by salespersons to advertise the product.

A *task* is an activity that is carried out by a user on an (abstract) object, e.g. *enter something* or *print something*. Therefore we understand tasks as *generic processes* that are instantiated on different kind of objects. In the OO world tasks are crosscutting concerns: While providing very good support for *object similarities*, OO does not help very much in design of *process similarities* on otherwise unrelated objects – and this leads to crosscutting. However, from a users' point of view, it is important that similar processes (e.g. entering or printing some kind of data) are represented similarly in the application's user interface: They should offer a similar 'look & feel'. As a result, we understand tasks as a concern of usability assurance.

Figure 6 shows the secondary dimensions <Features> and <Tasks> with some concerns. (Please do still ignore the different levels of grey.) As secondary dimensions are not meta-modeled, integrating them into the concern space is quite easy. We just have to assign the already existing units also to each secondary concern they address. Here we assigned to a feature concern exactly those use-case units which model a part or a refinement of the feature request. Analogously, we assigned all class units which are affected by a specific generic process to the corresponding task concern.Selection of a Concern-specific Slice

The concern space is now an integrated, multi-paradigm design description of our inventory system, built by integration of two artifact-based primary dimensions and two additional secondary dimensions. (Figure 7 explains the general integration process by an activity diagram.)

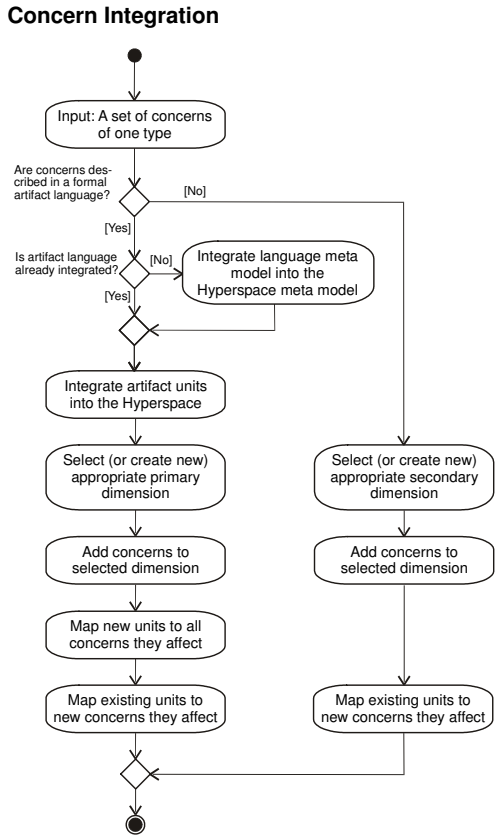

Figure 7 The process of concern integration

3.4. Selection of a Concern-specific Slice

We now use this integrated design description to select a concern-specific slice out of the whole system. By projecting this slice to the language-based dimensions, we then create concern specific artifact versions.

Consider we are using an iterative development process and want to build a minimal version of the inventory system as a first release. This minimal version should implement only the most important features, which in our case is just the one feature *Find data management*. Which classes and use-cases have to be taken into account to implement *Find data management*?

Because of the integrated design description of the system, we can find all relevant units by creation of a closure: Given a set C of concerns, we can compute the slice S implied by C (the closure of C) by determining the subgraph induced by all vertices (units and concerns) reachable from C via appropriate edges (relationships and concern mappings). This algorithm has to be instantiated appropriately according to the necessities of the respective artifact languages.

In Figure 6, the result of this algorithm is depicted in black color, while all units, concerns and relations that are omitted are colored grey. Figure 8 shows the effect of projecting the resulting slice on the <Use-cases> dimension and thus creating a concern specific version of the original use-case diagram. Analogously, a concern specific version of the original class diagram can be built.

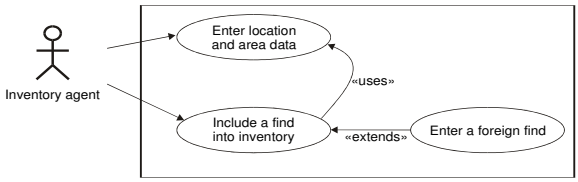

Figure 8 Projection of the concern-specific slice on the <Use-cases> dimension

4. Related Work

4.1. Hyper/J and the Hyperspace Approach

Hyper/J and the original Hyperspaces approach [3, 4, 6] can be seen as a specific instantiation of our model. Like our model, Hyper/J uses (implicitly) a meta-model (the meta-model of the Java language) and distinguishes two different types of dimensions. However, the meta-model of Hyper/J is hard-coded and not extendable. It is the base of the built-in <Classes> dimension, which is therefore the one and only primary dimension in Hyper/J. All additional dimensions introduced by the definition of hyperslices and hypermodules map to secondary dimensions in our model.

In the original Hyperspaces model, the mapping from units to concerns is realized by a *concern-matrix* that maps each unit to *exactly one* concern in *each* dimension. However, the implication that a single unit never addresses more than one concern of a dimension does not hold for the less formal languages as used in software design[6]. For that reason, we gave up the idea of a matrix-like n-dimensional concern space spanned by n axes, in favor of n *clusters* of concerns. This metaphor allows assigning units to *zero or more* concerns in each dimension by relations. This structure (which is perfectly represented by a graph) allows an easy definition of specialized systems which support only some of the described concerns by a transitive closure approach.

Another difference between the original model and our work is that we are not distinguishing between *declaration units* and *definition units*. Most design languages do not utilize the concepts of declaration and definition; units are typically introduced into the model by just naming them. Furthermore, in our model a unit can be mapped to more than one concern; thus additional declaration units are even formally not necessary. Of course, the abandonment of declarations leads to the danger that the transitive closure approach works too greedy and includes too many units while extracting a concern specific slice. However, using a graph-based representation, it is not difficult to control the creation of the transitive closure with additional constraints, e.g. based on regular path expressions [11].

4.2. Approaches based on UML design integration

In [7] the authors propose *UMLAUT*, a generic UML model and schema transformation framework [8], as a methodological base for aspect-oriented design with UML models. Like our

[6] Actors in use-case diagrams, like in Figure 4, are a good example for this. Typically, an actor relates to more than one use-case. Therefore, an actor may map to more than one concern on the <Use-cases> dimension, as shown in Figure 6.

model, their approach is based on an integrated meta-model of artifact languages. Aspect weaving and extraction of task-specific views are then driven by an extensible set of transformation rules and UML tagged attributes.

ConcernBASE [9] is an approach for a UML based framework for describing software architectures. It supports MDSoC on high-level software architectures by decomposing the system into different architectural concern spaces and definition of architectural views that represent single dimensions of these concern spaces.

A more component-related approach for aspect-oriented design and architecture is presented in [10]. The authors divide a complex design not by dimensions, but by functional domains into possibly non-orthogonal aspects, for which an optimal local design may be developed first. Constraints for interactions and connections between the local designs are formulated by contracts that are later used for automatic composition and detection of conflicts.

5. Conclusion

We introduced an approach to MDSoC which generalizes the Hyperspace approach of TARR and OSSHER. We gave up the idea of a matrix-like n-dimensional concern space spanned by n axes in favor of n clusters of concerns. This metaphor allows assigning units of artifact languages to more than one concern in each dimension. This property is not given if the metaphor of strict orthogonality is used.

We showed by an example how the Hyperspace approach is generalized to an arbitrary number of (artifact-based) primary dimensions using meta-model-based integration of artifact languages.

We believe that the strong connection between the dimensional structure of the hyperspace and the artifact languages is a good basis for an assessment of the appropriateness of design description languages. Ideally, each artifact language should define only one dimension, if a maximal separation of concerns is to be achieved. A language defining two kinds of concerns may not separate them well enough. Each secondary dimension may be an indicator of the absence of an additional language that might be useful.

The graph view of the Hyperspace allows an easy definition of specialized systems which support only some of the described concerns by a transitive closure approach. The graph-interpretation of meta-models is furthermore a good foundation for developing graph-based tools which support this approach to multidimensional separation of concerns in software development environments.

6. References

[1] **Rich Hilliard:** *Aspects, Concerns, Subjects, Views, … *.* In Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´99). Workshop: Advanced Separation of Concerns, 1999

[2] **Peri Tarr, Harold Ossher, William Harrison, Stanley M. Sutton Jr.:** *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In Proceedings of the 21st International Conference on Software Engineering (ICSE ´99), pp. 107-119, May 1999

[3] **Harold Ossher, Peri Tarr:** *Multi-Dimensional Separation of Concerns in Hyperspace.* Research Report RC21452(96717)16APR99, IBM Research Division, Almaden, April 1999

[4] **Harold Ossher, Peri Tarr:** *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000

[5] **Jürgen Ebert, Andreas Winter, Peter Dahm, Angelika Franzke, Roger Süttenbach:** *Graph Based Modeling and Implementation with EER/GRAL.* In Proceedings of the 15th International Conference on Conceptual Modeling (ER ´96). Lecture Notes in Computer Science 1157, pp. 163-178, 1996 (extended version: Report 11/1996, Universität Koblenz-Landau, Fachberichte Informatik)

[6] **Harold Ossher, Peri Tarr:** *Hyper/J: multi-dimensional separation of concerns for Java.* In Proceedings of the 22nd International Conference on Software Engineering (ICSE ´00), pp. 734-737, 2000

[7] **Wai-Ming Ho, François Pennaneac'h, Noël Plouzeau:** *UMLAUT: A Framework for Weaving UML-based Aspect-Oriented Designs.* In Technology of Object-Oriented Languages and Systems (TOOLS Europe) 30, pp 324-334, IEEE, June 2000

[8] **Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, François Pennaneac'h:** *UMLAUT: an extendible UML transformation framework.* In Proceedings of the 14th IEEE International Conference on Automated Software Engeneering (ASE ´99), IEEE, 1999

[9] **Mohamed Mancona Kandé, Alfred Strohmeier:** *On the Role of Multi-Dimensional Separation of Concerns in Software Architecture.* Position Paper for the 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000). Workshop: Advanced Separation of Concerns, 2000

[10] **Holger Giese, Alexander Vilbig:** *Towards Aspect-oriented Design and Architecture.* In Proceedings of the 15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ´00). Workshop: Advanced Separation of Concerns, 2000

[11] **Jürgen Ebert, Angelika Franzke:** *A Declarative Approach to Graph Based Modeling.* In Ernst W. Mayr, Gunther Schmidt, Gottfried Tinhofer (Eds.): Graphtheoretic Concepts in Computer Science, Lecture Notes in Computer Science 903, pp. 38-50, 1995