A Prototype Implementation of an Orthographic Software Modeling Environment

Colin Atkinson
University of Mannheim, Germany
atkinson@informatik.uni-mannheim.de

Dietmar Stoll
University of Mannheim, Germany
stoll@informatik.uni-mannheim.de

Christian Tunjic
University of Mannheim, Germany
tunjic@informatik.uni-mannheim.de

Jacques Robin
Universidade Federal de Pernambuco, Recife, Brasil
jr@cin.ufpe.br

ABSTRACT

Orthographic Software Modeling (OSM) is a view-centric software engineering approach that aims to leverage the orthographic projection metaphor used in the visualization of physical objects to visualize software systems. Although the general concept of OSM does not prescribe specific sets of views, a concrete OSM environment has to be specific about the particular views to be used in a particular project. At the University of Mannheim we are developing a prototype OSM environment, nAOMi, that supports the views defined by the KobrA 2.0 method, a version of KobrA adapted for OSM. In this paper we provide an overview of the KobrA 2.0 metamodel underpinning nAOMi and give a small example of its use to model a software system.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*

Keywords

Orthographic Software Modeling, View-based Modeling

1. INTRODUCTION

Orthographic Software Modeling (OSM) is based on three fundamental hypotheses — (a) that it is feasible to integrate the many different kinds of artifacts used in contemporary software engineering methods within a single coherent methodology in which they are treated as views, (b) that it is feasible to create an efficient and scalable way of supporting these views by generating them dynamically, on-the-fly, from a Single Underlying Model (SUM) using model-based transformations and (c) that it is feasible to provide an intuitive metaphor for navigating around these many views by adapting the orthographic projection technique underpinning the CAD tools used in other engineering disciplines.

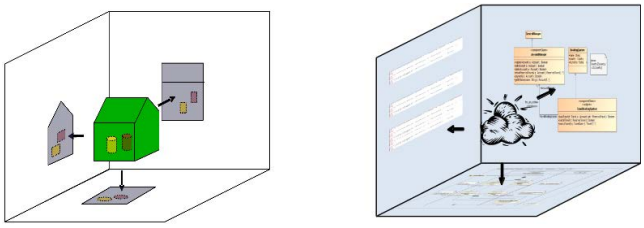

Figure 1: Orthographic Projection.

As shown in Figure 1, the main advantages of using the idea of orthographic projection to define the views used to visualize and described a system are that they (a) can be organized according to a simple and easy-to-understand metaphor and (b) collectively represent all the properties of a system with minimal overlap and redundancy. In practice this translates into a set of "dimensions", each containing well defined choices (or so called "dimension elements") that can be used to select individuals views.

As shown in Figure 2, the main advantage of making the artifacts used to describe a software system views of a SUM is that the number of pairwise coherence relationships that have to be maintained is reduced and new views can be introduced by simply defining their relationship to the SUM. Moreover, the importance of this advantage grows quickly as the size of the system and the complexity of the deployed development methodology increase. Another important advantage is that the dominance of one particular kind of view over the development process (e.g. code) at the expense of other kinds of views (e.g. graphical models) is reduced so that any appropriate type of views can be used to enrich the underlying description of the system, depending on the needs and skills of the stakeholder involved. This makes it possible to subsume all view types under the same, overarch-

VAO '13, July 2, 2013, Montpellier, France

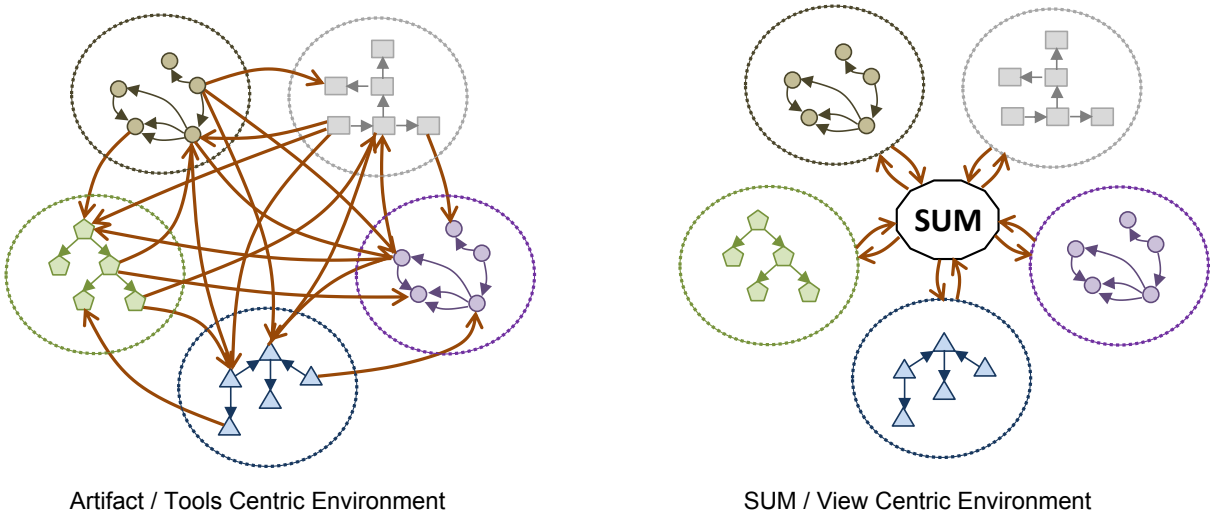

Figure 2: Consistency Dependencies in Artifact-oriented versus View-oriented Environments.

ing development process and methodology (e.g. agile-driven, focusing on small development cycles, or model-driven development, based on transformations between abstraction levels). Although the details of how the views are created from the SUM and how the SUM is updated from the views are not central to the approach, a natural implementation is to use the visualization and transformation technologies offered by model driven software engineering (MDSE).

To explore the validity of these hypotheses at the University of Mannheim we have been developing a prototype OSM modeling environment based on an enhanced version of the KobrA method for model-driven, component-oriented development, KobrA 2.0 [1]. This was chosen as a basis for the prototype, known as the *Open, Adaptable, Orthographic Modeling Environment* (nAOMi) [13] because its views were designed with the precise goals of being (a) genuine projections of a subject containing carefully selected subsets of information about that subject, (b) minimalistic in the sense that they should overlap to the smallest extent possible and contain the minimum necessary models elements, and (c) selectable via a set of independent "dimensions" which reflect different fundamental concerns of development (i.e. abstraction levels, composition or variants). In other words, KobrA already provided one of the "most orthogonal" sets of views for visualizing software systems of any contemporary method. More details about the actual views and dimensions defined in KobrA are presented in the following sections. More information on OSM can be found in [2] and [3].

nAOMi is implemented as an Eclipse plugin using the Eclipse Modeling Framework (EMF) as the underlying modeling platform and UML 2.0 tools [4] to generate and edit views. The KobrA 2.0 metamodel on which the current version of nAOMi is based is a specialization of the UML metamodel composed of three separate packages — one for the SUM, one for the views and one for the transformations (Figure 3). The UML was chosen as the base language because of its maturity and widespread acceptance, making the environment usable to the largest possible body of developers. UML elements not needed in KobrA 2.0 are excluded using OCL constraints while new elements or properties are

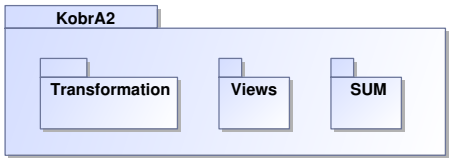

Figure 3: KobrA 2.0 Top Level Packages.

introduced by specializing existing elements.

The unique contribution of this paper is to elaborate on the structure of the KobrA 2.0 metamodel and how it is used to drive nAOMi. The three following sections each focus on one of the three main components of the metamodel — the SUM, the views and the transformations . This is followed by a brief overview of the OSM navigation paradigm in Section 5 before a small example of the approach is presented in Section 6. Section 7 then concludes the paper with related and future work.

2. SUM PACKAGE

Figure 4 depicts the internal structure of the *SUM* package which is based on the UML metamodel. There are three main subpackages, two containing the structural and behavioral constructs respectively, and one containing the constraints that ensure that the metaclasses are used according to the KobrA conventions and rules.

The *Classes* subpackage of the *Structure* package contains some of the most fundamental elements of the KobrA metamodel, such as *Class* and *ComponentClass*. The internal structure of this package is illustrated in Figure 5. *ComponentClass* represents objects with complex and reusable behaviors, while *Class* captures simple "data type" objects that have only very simple or non-reusable behaviors. The modeler has to decide whether it is necessary to model a specific part of the system as a *ComponentClass* and include state charts and activity diagrams, or whether it is sufficient to use a *Class* (which is limited to using OCL constraints).

ComponentClass inherits (indirectly via *Class*) from *Communications* so it also has the *isActive* attribute. This makes

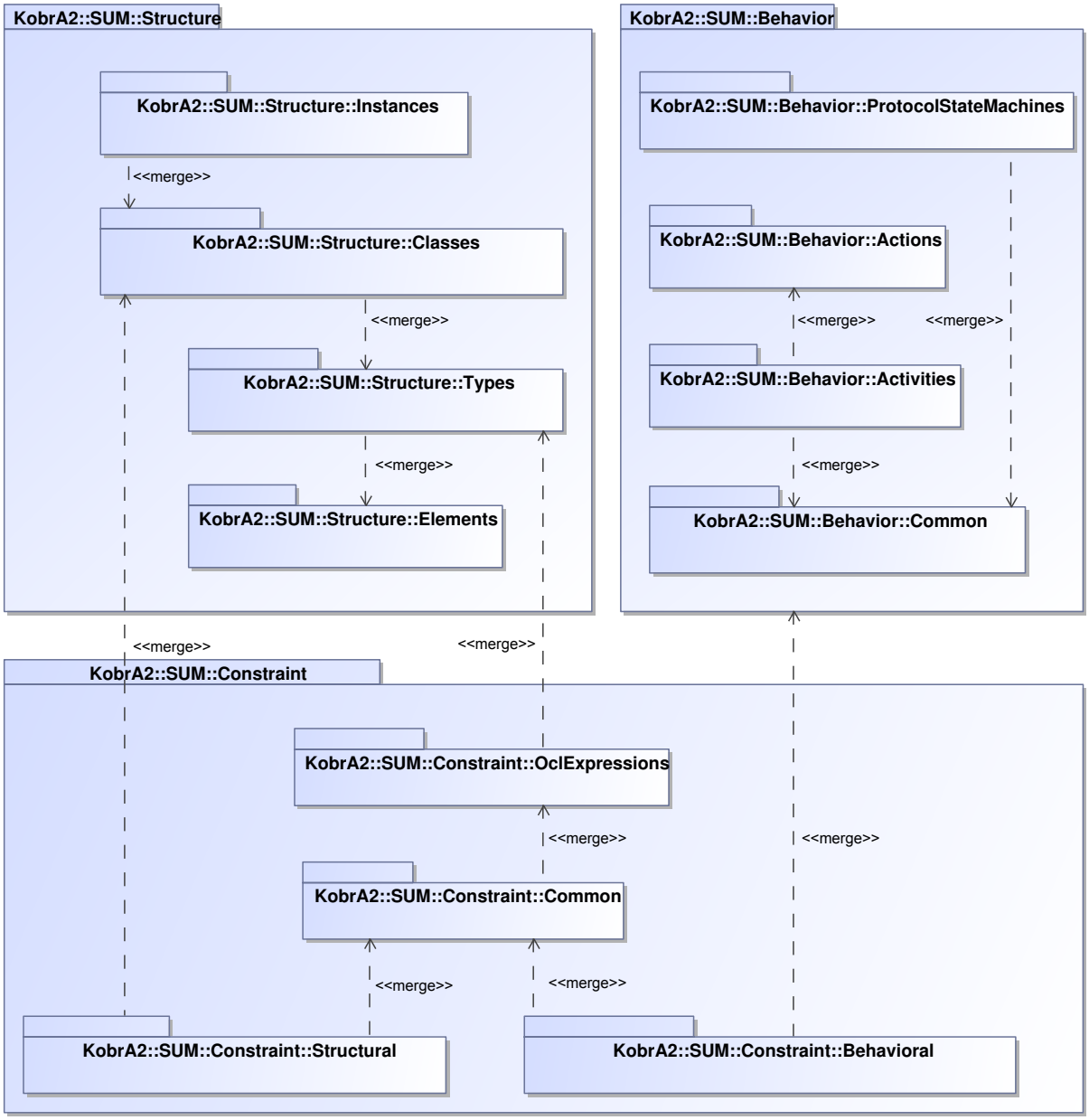

Figure 4: KobrA 2.0 SUM Package.

it possible to model whether its instances are active or passive. Active objects, which can be used to model threads and processes ([8] p. 438), start to execute their behavior as soon as they are created and perform operations spontaneously.

A *ComponentClass* may exhibit complex behavior. In KobrA, this behavior may be specified in the form of *UML State Diagrams* (defining acceptable operation invocation sequences), and in the form of *Activities* (defining algorithms of operations). UML *Interaction* elements (in sequence diagrams) can be derived from the activity elements and thus are not included in the SUM. As KobrA aims to facilitate automatic checking of allowed sequences of operation calls, *Protocol State Machines* are supported instead of general state machines. Since the latter include a large variety of elements not needed for specifying acceptable operation sequences or automatic checking, OCL constraints are used to prohibit the use of unwanted features.

```
context ComponentClass
-- only allow Activity elements or
     ProtocolStateMachines
inv: ownedBehavior->forAll(oclIsKindOf(Actitivity) or
      oclIsKindOf(ProtocolStateMachine))
```

For example, since KobrA has no concept of roles for components, the use of *role* also needs to be prohibited. The *part* association refers to owned properties of components whose attribute *isComposite* is true. As KobrA uses associations like *nests* and *creates* for components, *part*, *required* and *provided* are not needed. *Connectors* (i.e. *delegation* and *assembly*) are not used in KobrA either so *ownedConnector* is excluded.

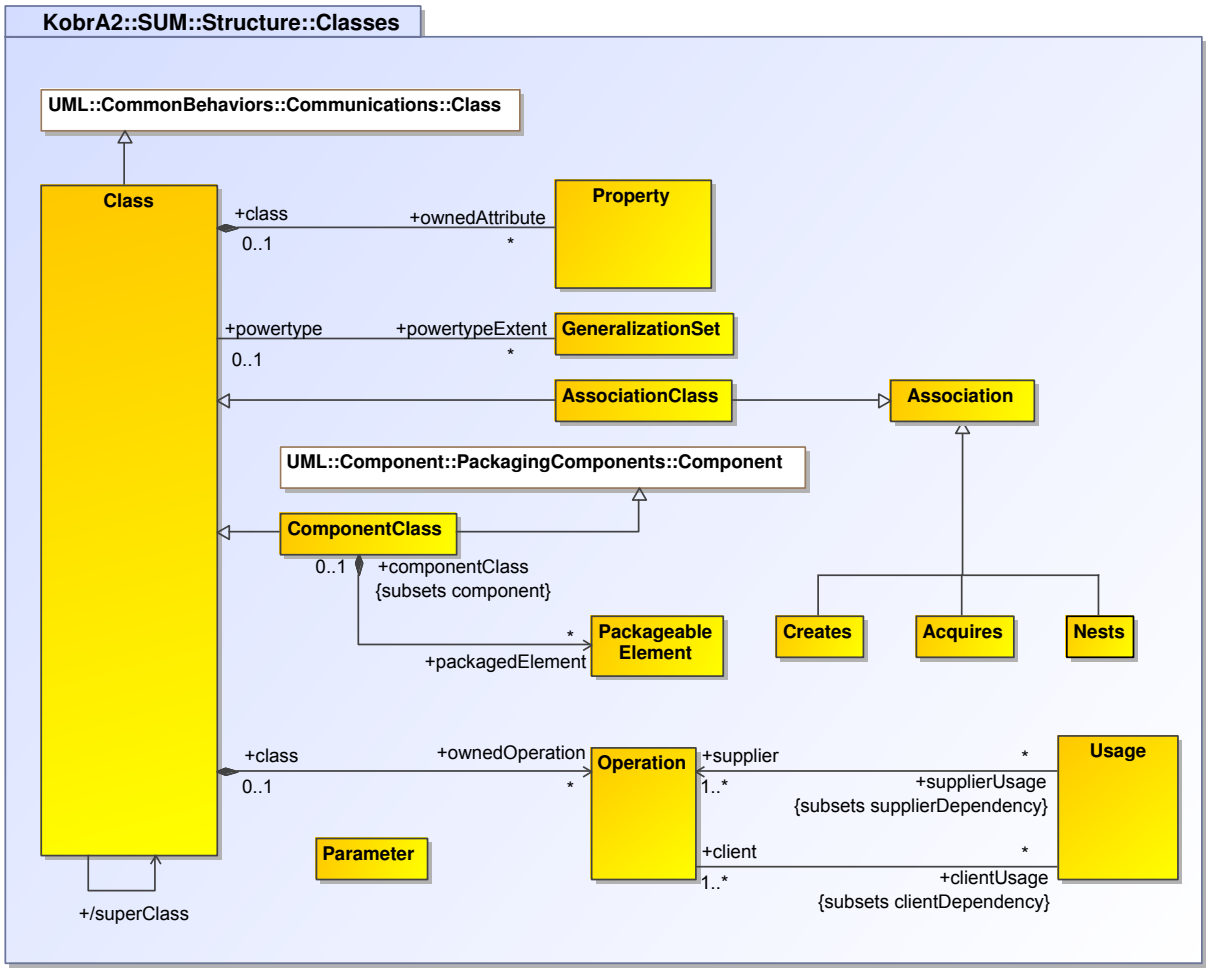

Figure 5: KobrA 2.0 Classes Package.

```
context ComponentClass
inv: role->union(part)->union(ownedConnector)
->union(collaborationUse)-> union(representation)
->union(realization)->union(required)
->union(provided)->isEmpty()
```

3. VIEWS PACKAGE

The structure of the *Views* package is illustrated in Figure 6. Again, since most of the views defined in KobrA 2.0 are based on UML diagrams, the view metamodels have similar elements to the SUM metamodel. The big difference to the SUM is that there are no restrictions on the use of the view metamodel elements. For instance, views for a particular purpose such as supporting model checkers can be supported by adding elements unrelated to the UML.

The substructure of the *Views* package reflects the types and organization of the KobrA views according to the view "dimensions" supported in nAOMi (cf. example in Section 6). At the top level, the *Views* package is thus decomposed into the *Specification* and *Realization* options of the encapsulation dimension. These, in turn are both decomposed into the *Structural*, *Behavioral* and *Operational* options of the Projection dimension. Finally, with the exception of the behavioral option, these are also all subdivided into the *Service* and *Type* options of the granularity dimension. This dimension, with its two options, is an addition to the original version of KobrA.

The *Service* view shows the direct, publicly visible relationships of the subject *ComponentClass* to other *ComponentClasses*, while the *Type* view shows the publicly visible relationships of the subject to simple *Classes*. As with the SUM, constraints have been defined to control what can go into each view and when they are well formed. For every view, a constraint enumerates all allowed elements (not shown in this paper).

In the following, some of the other constraints for the *Service* view are elaborated. Since this view is a black-box view, the internals of *ComponentClasses* (*nestedClassifier*) are not shown.

```
context ComponentClass
-- no nested classifiers, no protocol
inv: nestedClassifier->union(protocol)->isEmpty()
```

Classes are only allowed if they are generalizations of *ComponentClasses*, (or any of its superclasses, since a *ComponentClass* may inherit from a class as shown in the constraints with context Class. The following invariants ensure that only publicly visible attributes and operations are in this view, for both classes and *ComponentClasses* (which inherit from *Class*).

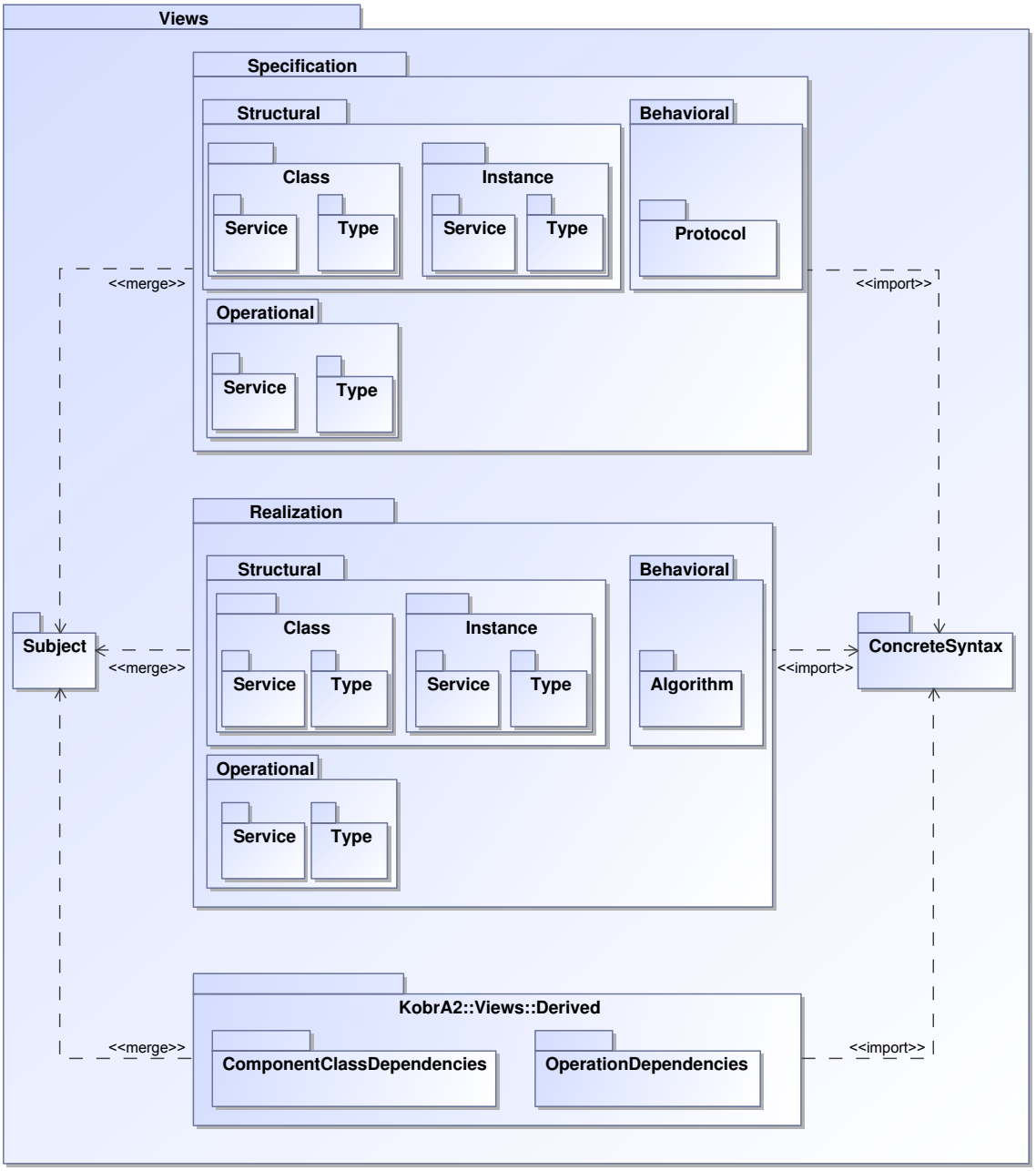

Figure 6: KobrA 2.0 Views package nesting.

```
context Class
-- only allow classes that are direct or indirect
     generalizations of ComponentClasses in this view
def: ccGeneralization : generalization.specific->
     exists(oclIsKindOf(ComponentClass))
inv:  generalization.specific->select(oclIsTypeOf(
     Class))->exists(s|s.ccGeneralization)
       or ccGeneralization
-- only public attributes in this view
inv: ownedAttribute->forAll(visibility=#public)
-- only public Operations are allowed in the
     specification
inv: ownedOperation->forAll(visibility=#public)
```

Only operation signatures are shown in this view, so pre-, post- and bodyconditions, as well as activities are omitted, which is reflected in the last constraint.

```
context Operation
-- only the signature of the Operation is shown, not
     its behavior (role name "method" refers to the
     Activities of the operation), or dependencies
inv: method->union(precondition)->union(body)->union(
     postcondition)->isEmpty()
```

4. TRANSFORMATIONS PACKAGE

The package *AllViews* provides the foundation for specifying the transformations between the SUM and the views in both directions. Part of the package's contents are shown in Figure 7. The *Abstraction* concept (which is in fact a

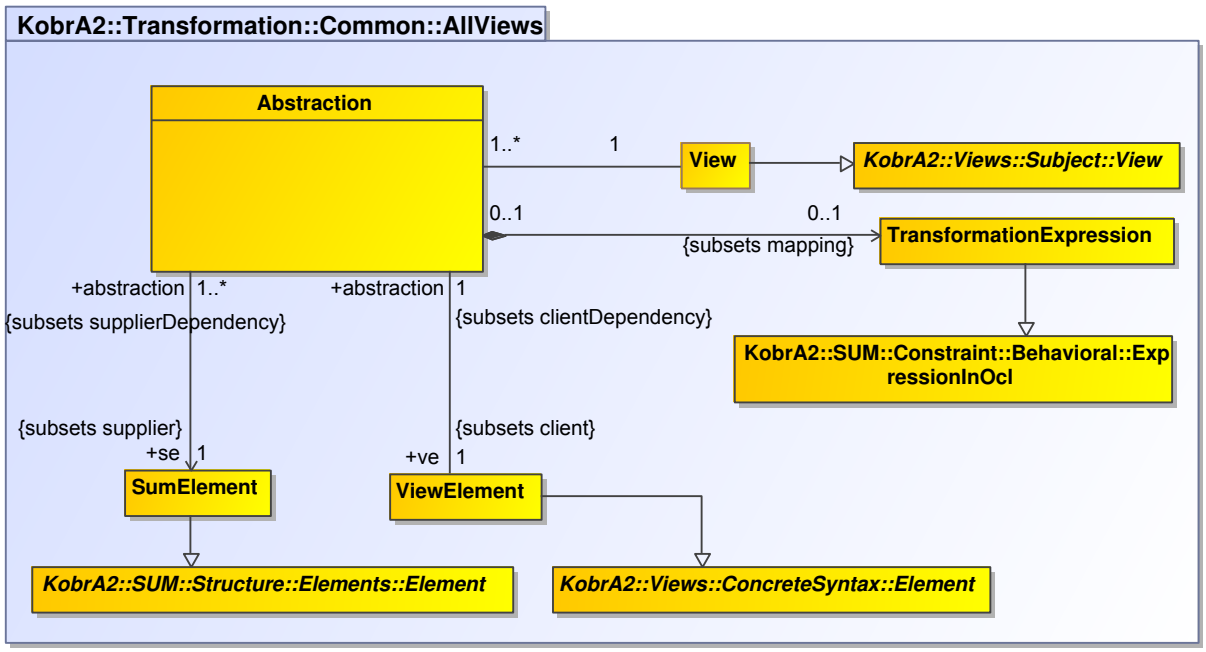

Figure 7: Transformation abstractions.

dependency reused from the UML but with additional constraints) plays the key role in relating elements from the SUM to elements of a view. *Abstraction* is actually *mapped* to *ExpressionInOcl*. When appearing in transformations, the equals sign links elements in the SUM to the respective elements in the view, and vice versa. For instance, equality of the *general* meta-association of a *Generalization* in a transformation invariant means that, when following *general*, there must be an element in the SUM and in the view for which similar transformation expressions are specified. In the case of KobrA 2.0, which has many projections that just select a subset of elements using one-to-one abstractions, this allows concise declarative *TransformationExpressions*. Together with the view constraints, a CASE tool can be implemented which uses a transformation language of the implementor's choice, for instance the Atlas Transformation Language (ATL) [11] or QVT [9]. The role names *se* and *ve* are short for *SumElement* and *ViewElement*, respectively. These roles subset the *client* and *supplier* roles from the UML.

SUM elements are translated into UML elements with stereotypes, so that the views are easy to manage for developers familiar with the UML. The bidirectional mappings between stereotyped view elements and non-stereotyped SUM elements are expressed in the constraints of the *AssociationAbstraction*, a subclass of the *Abstraction* from the *AllViews* package. This is also an example of a transformation which is reused in other views.

```
context AssociationAbstraction
inv: ve.memberEnd = se.memberEnd
inv: ve.ownedEnd = se.ownedEnd
ivn: ve.navigableOwnedEnd = se.navigableOwnedEnd
inv: se.oclIsKindOf(Acquires) implies ve.
     hasStereotype('acquires')
inv: ve.hasStereotype('acquires') implies se.
     oclIsKindOf(Aquires)
inv: se.oclIsKindOf(Nests) implies ve.hasStereotype('
     nests')
inv: ve.hasStereotype('nests') implies se.oclIsKindOf
     (Nests)
inv: se.oclIsKindOf(Creates) implies ve.hasStereotype
     ('creates')
inv: ve.hasStereotype('creates') implies se.
     oclIsKindOf(Creates)
```

Figure 8 shows the main elements involved in the transformation of the black box structural view for *ComponentClasses*. The first transformation constraint is on the view and declares the starting point for the transformation. It states that the *subject ComponentClass* and its generalizations (using a SUM utility function, *superClosure*) are in the view.

The following transformation rules illustrate how to create the output (i.e. view) elements from the input (i.e. SUM) elements, such as the publicly visible attributes and operations of the *ComponentClass* and the *acquired ComponentClasses*. The first constraint for *ComponentClassAbstraction* states that references to potential general classes (and *ComponentClasses*) of *ComponentClasses* are mirrored in the view. In addition, *ComponentClasses* will be shown with the corresponding stereotypes. The *ComponentClass* owns various types of associations, so in this view only the *acquires* associations are selected (whose transformation rules are covered in the common transformation packages).For classes and *ComponentClasses*, only publicly visible attributes and operations appear in the view. Class invariants are also copied. Classes that may appear in this view (e.g. as generalizations of *ComponentClasses*) may have a powertype (role name *powertypeExtent*) which will be displayed.

The last transformation statement copies the *class* references of operations. As with all views, the transformation rules, the common transformation statements (which also cover operations) and the view constraints serve as a specification for the implementation of a view. Individual CASE tools can use different implementation techniques as long as they conform to the semantics of these rules and constraints.

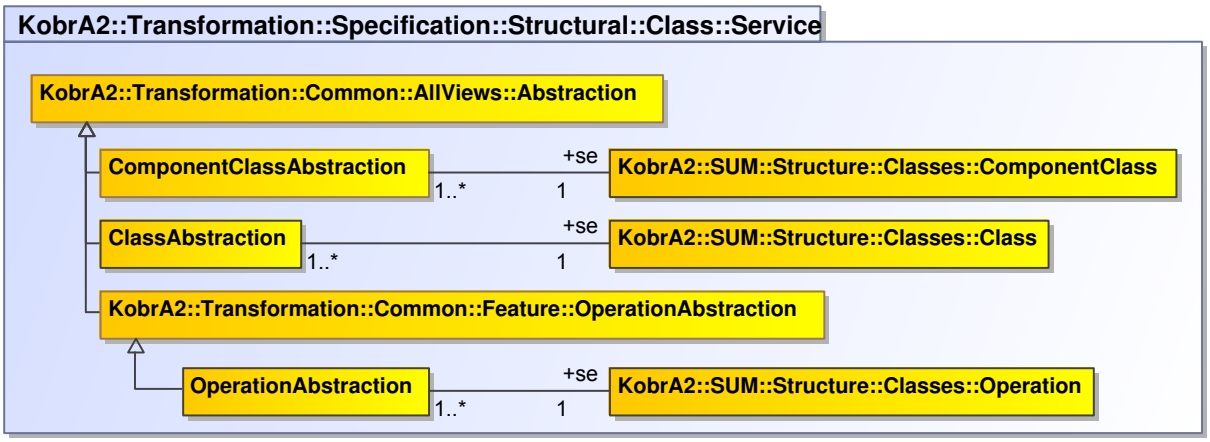

Figure 8: Transformation to the Specification Structural Service View.

```
context KobrA2::Views::Subject::
    SpecificationStructuralClassService
inv: ownedMember->select(oclIsKindOf(Class)) =
    subject.superClosure->union(subject.acquires.
    superClosure)

context ComponentClassAbstraction
inv: ve.superClass = se.superClass
inv: ve.hasStereotype('ComponentClass')
inv: se.isSubject implies (ve.hasStereotype('subject
    ') and ve.ownedMember->select(oclIsKindOf(
    Association)) = se.ownedMember->select(
    oclIsKindOf(Acquires)))

context ClassAbstraction
inv: ve.ownedAttribute = se.ownedAttribute->select(
    visibility=#public)
inv: ve.ownedOperation= se.ownedOperation->select(
    visibility=#public)
inv: ve.'inv' = se.'inv'
-- copy powertypeExtent that is only allowed for
    class
inv: ve.powertypeExtent = se.powertypeExtent

context OperationAbstraction
inv: ve.class = se.class
```

For the black box type view, only publicly visible attributes and operations of classes (as opposed to *ComponentClasses*) used by the *subject* can be seen. This is specified in the first rule which defines owned members of the view and thus serves as the starting point of the transformation. *cbbTypes* is a utility function defined in the SUM which computes the black box types by selecting the types of the subject's public attributes and parameter types of its public operations.

Class invariants and potential powertypes and connections to the classes in this view are shown as well. There may also be *Enumerations*, for which the *EnumerationLiterals* are displayed.

The transformation rules for this view are almost the same as the realization transformation constraints from the package *Transformation::Realization::Structural::Class::Type*. The differences are the *select(visibility=#public)* statements for operations and attributes.

```
context KobrA2::Views::Subject::
    SpecificationStructuralClassType
inv: ownedMember->select(oclIsKindOf(Class) or
    oclIsKindOf('Enumeration') or oclIsKindOf(
    Association)) = subject->union(subject.cbbTypes)

context ComponentClassAbstraction
inv: se.isSubject implies ve.hasStereotype('subject')

context ClassAbstraction
inv: not se.oclIsKindOf(ComponentClass) implies (
    ve.ownedAttribute = se.ownedAttribute->select(
        visibility=#public)
    ve.ownedOperation = se.ownedOperation->select(
        visibility=#public))
inv: ve.powertypeExtent = se.powertypeExtent
inv: ve.superClass = se.superClass
inv: 've.inv' = 'se.inv'

context ComponentClassAbstraction
inv: se.isSubject implies ve.hasStereotype('subject')

context EnumerationAbstraction
inv: ve.ownedLiteral = se.ownedLiteral

context EnumerationLiteralAbstraction
inv: ve.specification = se.specification.
    stringInSignature
```

5. NAVIGATION

Most of today's tools use some combination of trees to organize the content of models as well as the views used to visualize a software system or component. In an any environment incorporating a number of different tools there is invariably a large number of different trees storing a heterogeneous mix of artifacts including model elements (e.g. classes, instances, associations), diagrams (e.g. class diagrams, state diagrams) and other artifact types (source code, XML files, configuration files). To work with all the views in a traditional development environment, therefore, engineers typically have to learn about the organization structures of all the incorporated tools.

In contrast to conventional paradigms for organizing and navigating the many views used to visualize a system, OSM employs the metaphor of a multi-dimensional cube. More specifically, as illustrated in Figure 9, OSM regards dimension of the underlying methodology as representing a different dimension of the cube, and each independently variable aspect of that dimension is a selectable dimension element. Selecting a view thus simply corresponds to selecting a single cell within the cube. In general, three types of dimensions are supported: static dimensions in which the number of

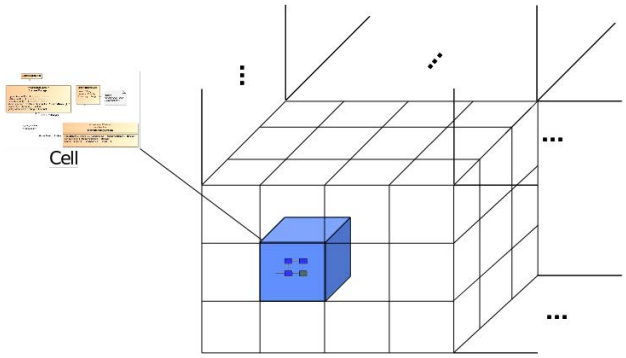

Figure 9: Dimension-based navigation.

selectable elements (i.e. coordinates) is fixed, dynamic dimensions in which the number of elements is dynamic (i.e. derived from the SUM), and mixed dimensions which have both static and dynamic elements.

To support the OSM dimension based navigation metaphor for KobrA, we defined the seven dimensions indicated on the left hand side of Figure 10 which is a sceenshot of nAOMI. The Abstraction dimension (not expanded here), which has three static dimension elements, PIM (platform independent model), PSM (platform specific model) and Code, captures the model-driven development concern of KobrA. The version dimension captures the state of the modeled system at specific points in time. The Component dimension, which has dynamic dimension elements defined by instances of the class ComponentClass in the SUM, captures the component-based development concern of KobrA.

The Encapsulation dimension, which has two fixed elements, supports the distinction between Specification (black box) and Realization (white box) views of components, while the Projection dimension with the fixed elements Structural, Operational and Behavioral covers the different information types. The Granularity dimension provides a finer grained distinction between views describing the types used by components (Type granularity) and views describing the required and provided interfaces (Service granularity). The Operation dimension allows a selection of individual operations.

In the ideal case, when all views are truly orthogonal, the choices that can be made in each dimensions are completely independent. However, this is very difficult to achieve in software engineering. The approach still works if the views are not completely orthogonal, but dependencies then occur between different choices in different dimensions, so that the decisions made in one dimensions may affect choices possible in another dimension. This is best handled by giving dimensions a precedence ranking determined by the order in which they appear (the top being the highest). When an element in a dimension is selected, the tool automatically makes default selections for dimensions of lower precedence (i.e. dimensions lower down) and disables selections that would navigate to cells (i.e. views) which are not (yet) defined by the method at hand.

6. SHOPPING CART EXAMPLE

To show how a software system can be specified using nAOMi, this section presents a case study based on a shopping cart system. A *ShoppingCart* component collects and

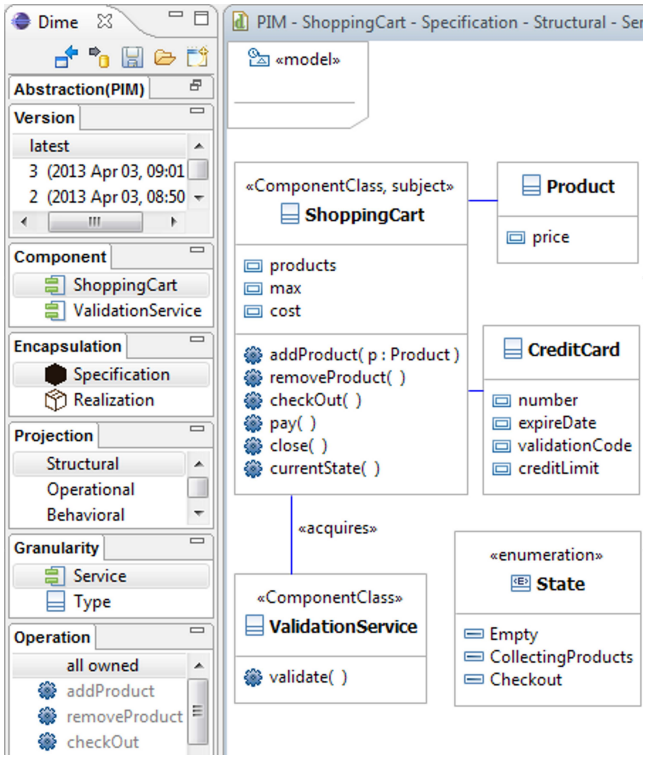

Figure 10: Specification Structural View.

manages the products selected by users and supports payment via a credit card. Figure 10 illustrates a structural view of the component.

In the dimension navigator on the left hand side, PIM was chosen for the "Abstraction Level" (not expanded in the screenshot). The second dimension is the state of the software system at a certain point in time. The picture shows that the latest available version was chosen. As with every choice in a dimension, it may influence the options in lower ranked dimensions. The component under consideration is the ShoppingCart, for which a black box view is selected in the next dimension. After the user selects the structural projection option and the service level granularity, the tool automatically chooses the option for all operations in the last dimension, as there is no editor registered for the other options.

The component under development is presented with the stereotype *subject* and its relationship to other components and classes is shown in the view, which corresponds to a cell of the multi-dimensional navigation cube, and is generated on-the-fly from the SUM when it is selected. The classes *Product* and *CreditCard* can be used as data types in the operations of the component.

Figure 11 illustrates the operational view in which an operation can be formalized using pre- and postconditions. The precondition corresponds to the *assumes* clause in and the postcondition corresponds to the *result* clause. As in the UML, the precondition of an operation must be true when the operation is invoked and the postcondition must be true when the operation is finished. The operation *addProduct* in Figure 11 must be in state *CollectingProducts* or *Empty* when invoked. This is also visible in the behavioral view,

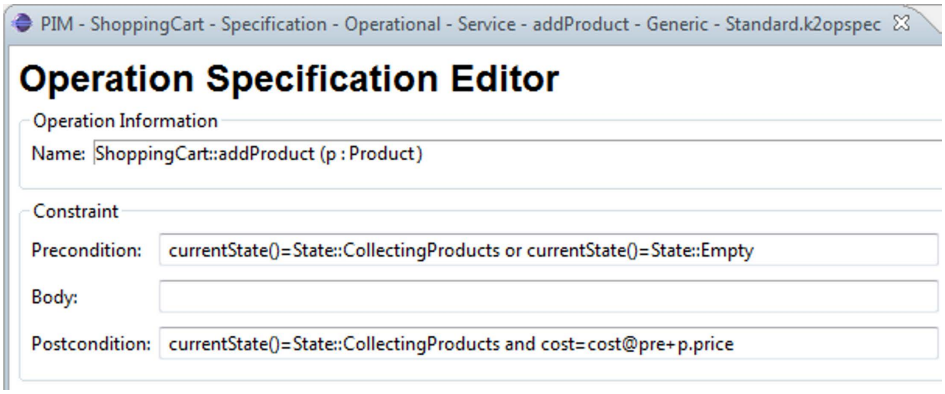

Figure 11: addProduct() Operation Specification.

since there are only two transitions with the operation *addProduct*. Both leads to the state *CollectingProducts* which is also a postcondition of the operation. The second postcondition is that the *cost* attribute of the component must be increased by the price of the added product. The pre- and postcondition can be expressed using the OCL. The properties of the component, states and operation parameters can be used to formalise the constraints like as in this example.

Figure 12 shows the publicly visible behaviour of the *ShoppingCart* component with states and transitions. The conditional transitions map to operations of the component. Like every view, this view is also synchronized with the SUM so that it is guaranteed that its operations, states and properties are consistent with those in the structural view.

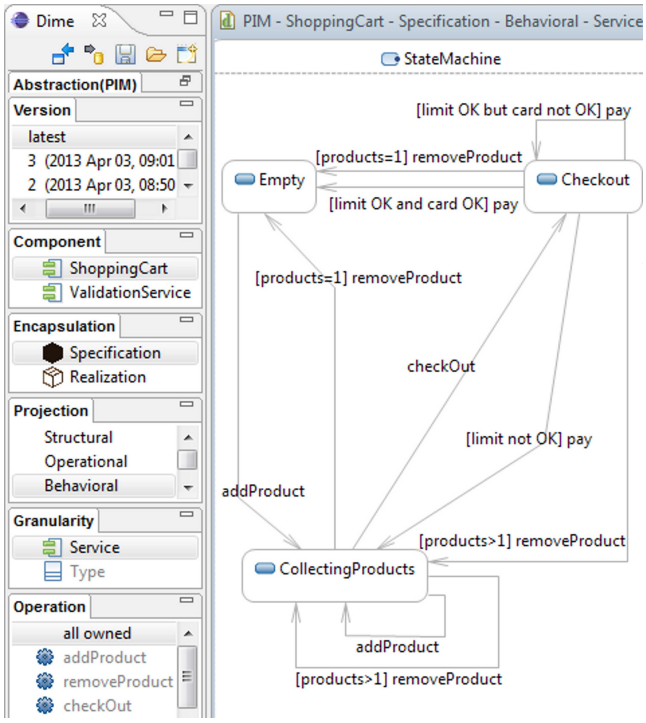

Figure 12: Specification Behavioral Model.

Although the operational view seems to be similar to the behavioral view because of the overlapping information within them, there are significant differences. The focus of the operational view is on a precise formal definition of an operation of a component. The operations can be enriched by pre- and postconditions which can be defined using complex OCL statements, that formalize the complete behavior of an operation. The additional information in the OCL statements can be used for code generation and documentation.

7. CONCLUSION

At the beginning of the paper we identified three fundamental hypothesis upon which the notion of OSM is based — (a) that it is feasible to integrate the many different kinds of artifacts used in contemporary software engineering methods within a single coherent methodology in which they are treated as views, (b) that it is feasible to create an efficient and scalable way of supporting these views by generating them dynamically, on-the-fly, from a Single Underlying Model (SUM) using model-based transformations and (c) that it is feasible to provide an intuitive metaphor for navigating around these many views by adapting the orthographic projection technique underpinning the CAD tools used in other engineering disciplines.

The prototype tool, nAOMi, described in this paper represents the first step towards demonstrating the validity of these hypotheses and showing that OSM is a viable approach to software engineering. Of the three hypotheses, (a) and (c) are most convincingly demonstrated by the prototype, since it shows that it is indeed possible to support all the views of the KobrA method within a single navigation metaphor. The prototype tool does not demonstrate the validity of hypothesis (b) to the same extent as the others due to its small size. Although it demonstrates the feasibility of generating views from the SUM and vice-versa, the question of whether such an approach scales up to large environments is still open.

Although nOAMi is the only tool developed with the specific aim of supporting KobrA-based OSM, several other tools and methods have similar properties or aims. For example, Glinz et al. [10] describe a tool with a fisheye zooming algorithm which lets the user view a model with varying amounts of detail depending on the context. It has to be investigated whether it is possible to combine the fisheye zooming concept with the dimension-based navigation paradigm. While the KobrA 2.0 implementation of nAOMi heavily uses UML diagrams for developers, Glinz et al. use

custom diagram types, e.g. for structural and behavioral views.

An approach which also emphasizes the description of formal consistency rules (*correspondences*) between views is RM-ODP [5][6]. However, this approach does not explicitly mention the notion of a SUM and thus implies that consistency rules should be defined in a pairwise fashion between individual pairs of views. ArchiMate [7], which complements TOGAF [12], is an enterprise architecture modeling language which offers two orthogonal "dimensions" for modeling, (*business*, *architecture*, and *technology*) layers and (*informational*, *behavioral* and *structural*) aspects and also suggests two more dimensions, *purpose* and *abstraction level*. However, as many of these views span multiple choices of a single "dimension", the intuitive dimension-based navigation metaphor of OSM can not be easily applied. There are also more general approaches for view-based modeling but they are less specific in terms of consistency rules between views and provide little guidance on how to manage and navigate views, for example the Zachman Framework [14].

Regarding the practical use of OSM environments in the future, the biggest challenge is developing appropriate SUM metamodels which can accommodate all the types of views and services that software engineers are accustomed to today. For this first prototypical SUM-based environment supporting the OSM approach we had a method at our disposal (KobrA) that already defined a full set of orthogonal UML-based views. This allowed us to model the required SUM and view metamodels by simply adapting the UML metamodels, removing and adding model elements as needed.

In doing so we were able to manually ensure that the metamodels fulfilled the two core requirements of SUM-based environments — (1) being minimalistic and (2) redundancy free. If SUM-based software engineering environments are to take off, and to be introduced into existing, heterogeneous environments, more sophisticated ways of integrating existing metamodels into a single unified metamodel will be required.

8. REFERENCES

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-Based Product Line Engineering with UML*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, November 2001.

[2] C. Atkinson, D. Stoll, and P. Bostan. Orthographic Software Modeling: A Practical Approach to View-Based Development. In *Evaluation of Novel Approaches to Software Engineering*, volume 69 of *Communications in Computer and Information Science*, pages 206–219. Springer Berlin Heidelberg, 2010.

[3] C. Atkinson, D. Stoll, and C. Tunjic. Orthographic Service Modeling. In *Proceedings of 15th IEEE EDOC Conference Workshops (EDOCW), Helsinki, Finland*, 2011.

[4] Eclipse Foundation. UML2Tools. `http://wiki.eclipse.org/MDT-UML2Tools`, 2013.

[5] ISO/IEC and ITU-T. The Reference Model of Open Distributed Processing. RM-ODP, ITU-T Rec. X.901-X.904 / ISO/IEC 10746. `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`, 1998.

[6] J. I. J. Jose Raul Romero and A. Vallecillo. Realizing Correspondences in MultiViewpoint Specifications. In *Proceedings of the Thirteenth IEEE International EDOC Conference, 1 - 4 September 2009, Auckland, New Zealand*, September 2009.

[7] M. Lankhorst. *Enterprise Architecture at Work*. Springer Berlin Heidelberg, 2009.

[8] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. `http://www.omg.org/cgi-bin/doc?formal/07-11-02`, November 2007.

[9] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0. `http://www.omg.org/spec/QVT/1.0/PDF/`, April 2008.

[10] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett. An effective layout adaptation technique for a graphical modeling tool. In *Proceedings of the 2003 International Conference on Software Engineering, Portland*, 2003.

[11] The Atlas Transformation Language (ATL). Official Website. `http://www.eclipse.org/atl/`, 2013.

[12] The Open Group. TOGAF Version 9 - The Open Group Architecture Framework. `http://www.opengroup.org/architecture/togaf9-doc/arch/index.html`, Feb 2009.

[13] University of Mannheim - Software Engineering Group. nAOMi - opeN, Adaptable, Orthographic Modeling EnvIronment. `http://eclipselabs.org/p/naomi`.

[14] J. A. Zachman. The Zachman Framework: A Primer for Enterprise Engineering and Manufacturing. `http://www.zachmaninternational.com`, 2009.