Towards a Tool-Oriented Taxonomy of View-Based Modelling

Thomas Goldschmidt[1], Steffen Becker[2], Erik Burger[3]

[1]ABB Corporate Research Germany, Industrial Software Systems Program
thomas.goldschmidt@de.abb.com
[2] University of Paderborn, steffen.becker@uni-paderborn.de
[3] Karlsruhe Institute of Technology (KIT), burger@kit.edu

Abstract: The separation of view and model is one of the key concepts of Model-Driven Engineering (MDE). Having different views on a central model helps modellers to focus on specific aspects. Approaches for the creation of Domain-Specific Modelling Languages (DSML) allow language engineers to define languages tailored for specific problems. To be able to build DSMLs that also benefit from view-based modelling a common understanding of the properties of both paradigms is required. However, research has not yet considered the combination of both paradigms, namely view-based domain specific modelling to a larger extent. Especially, a comprehensive analysis of a view's properties (e.g., partial, overlapping, editable, persistent, etc.) has not been conducted. Thus, it is also still unclear to which extent view-based modelling is understood by current DSML approaches and what a common understanding if this paradigm is. In this paper, we explore view-based modelling in a tool-oriented way. Furthermore, we analyse the properties of the view-based domain-specific modelling concept and provide a feature-based classification of these properties.

1 Introduction

Building views on models is one of the key concepts of conceptual modelling [RW05]. Different views present abstract concepts behind a model in a way that they can be understood and manipulated by different stakeholders. For example, in a component-based modelling environment, stakeholders, such as the system architect or the deployer will work on the same model but the system architect will work on the connections and interactions between components whereas the deployer will focus on a view showing the deployment of the components to different nodes [Szy02].

This is not only true for different types of models, as e.g., the different abstraction levels defined by the Model Driven Architecture (MDA) [MCF03], but also for having different views on the same models. Specialised views on a common underlying model foster the understanding and productivity [FKN+92] of model engineers. Recent work [ASB09] has even promoted view-based aspects of modelling as core paradigm.

Frameworks for the creation of Domain Specific Modelling Languages (DSMLs) allow to efficiently create tailored modelling languages. Different types of DSML creation approaches have emerged in recent years [KT08, CJKW07, MPS, Ecl11b, Ecl11a, KV10].

Many of these approaches implicitly allow for, or explicitly claim to, support the definition of views on models.

However, a comprehensive analysis of view-based aspects in DSML approaches has not been performed, yet. Furthermore, there is no clear determination on these concepts given in literature. Work on architectural, view-based modelling is mostly concerned with its conceptional aspects (e.g., [RW05, Cle03, Szy02, ISO11]). Their definitions are however on architecture level and do not deal with specific properties of views within modelling tools, such as their scope definition, representation, persistency or editability.

In order to agree on requirements for view-based modelling and to be able to decide which view-based DSML approach to use, language engineers and modellers require a clear and common understanding of such properties. Researchers have partially used these properties explicitly or implicitly in existing work (e.g., [GHZL06, KV10]). However, as these concepts and properties are scattered across publications and are often implicitly considered, this paper aims at organising them.

In order to get an overview on the view-based capabilities in existing DSML frameworks, we analysed a selection of DSML frameworks. We included graphical DSML frameworks ([KT08, CJKW07, Ecl11a]) as well as textual DSML frameworks ([MPS, Ecl11b, KV10]) to identify a common understanding of view-based domain-specific modelling.

The contribution of this paper is two-fold: First, we provide a common definition of view-based modelling from a tool oriented point of view. Second, we identify the different properties and features of view-based modelling in DSML approaches. The work presented in this paper is beneficial for different types of audience. Modellers can use the properties to make their requirements on view-based modelling more explicit. Tool builders can use the presented properties to classify and validate their approaches or as guidelines for the development of new view features. Researchers can benefit from the common definition of views on which further research can be based.

The remainder of this paper is structured as follows. Section 2 presents an overview as well as a differentiation of different notions of the term “view” that are used as categories for the classification scheme. Properties of view-types, views and specific editor capabilities are given in Section 3. Related work is analysed in Section 4. Section 5 concludes and outlines future work.

2 Determination of View-Points, View-Types, and Views

In this paper we try to clarify the understanding of the common terminology in view-based modelling. As our understanding of view, view points, view types and modelling originates from a tooling perspective, our understanding of the terms varies slightly from existing definitions like the ISO 42010:2011 standard [ISO11]. Therefore, in this section we are illustrating an example of a view-based modelling approach from our own industrial experience and illustrate using this example our understanding of the terms. We find the same understanding realised in many of the tools we have classified and surveyed in order to come up with this taxonomy. Finally, we discuss our terminology in the context of existing definitions like the ISO standard.

2.1 Tool basis

In orde to come up with a generic taxonomy for tools in the view-based modelling area we analysed several DSML tools. The selection process for the tools that we analysed was based on the following criteria:

1. The search was based on electronic databases, e.g., ACM DigitalLibrary IEEEXplore, SpringerLink as well as references given by DSML experts. The search strategy was based on the keywords "domain-specific", "modelling", "language", "view-based", "view-oriented", "views" and "framework" in various combinations. The search was conducted in several steps within December 2010 and January 2011.
2. Domain-specific modelling includes approaches stemming from several different research areas. Therefore, we included DSML approaches coming from different areas, e.g., meta-case tools, compiler-based language engineering as well as general model-driven engineering.
3. For being recognised as view-based DSML framework it should be possible to define new or use existing metamodels and create multiple concrete syntaxes for them.
4. Approaches for which a tool or framework was available were included. This ensured that approaches only having a theoretical or very prototypical character were excluded.
5. Approaches which have a tool which is not longer maintained or where the project was considered dead were excluded.
6. Finally, we excluded tools for which we found no indications for industrial relevance such as experience reports or real world evaluations. Thus, only tools proven to be mature enough for industrial application were included.

The selection process was very strict as our goal was to evaluate only those tools which had a chance of being employed in industrial projects. Of course, this may threat the general applicability of our taxonomy but on the other hand ensures that the taxonomy is applicable by industry. Finally the tools we analysed were the following: Eclipse GMF [Ecl11a], MetaEdit+ [KT08], Microsoft DSL tools [CJKW07], Jetbrains MPS [MPS] and Eclipse Xtext [Ecl11b]. Due to space restrictions, we cannot include the whole survey here; however, preliminary results are available online.[1]

2.2 Terminology

Figure 1 presents our example language and its views from the business information domain. The example language serves as a DSL to model business entities, their relations, their interactions and their persistence behaviour.

[1] http://sdqweb.ipd.kit.edu/burger/mod2012/

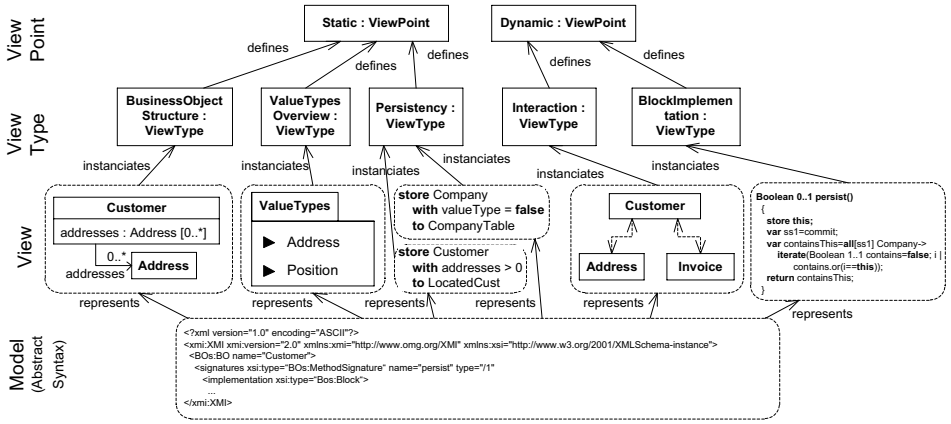


Figure 1: Example language and its viewpoints and viewtypes used by some example views

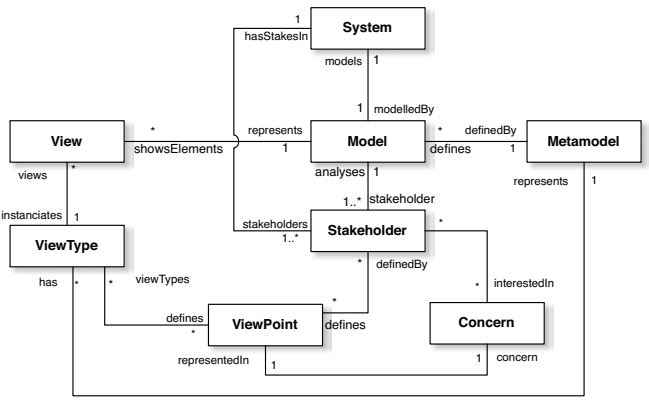


Figure 2: Terminology for view-based modelling used in this paper.

Our language therefore consists of two viewpoints: a static viewpoint to model the static structure of the business entities and a dynamic viewpoint to model their dynamics. The static viewpoint consists of three view types: a structure viewtype that defines how to presents the business objects and their relations in a class digram like notation, a value viewtype that defines how to shows the attributes of the business entities, and a persistency viewtype that defines how to link the business entities with the database and defines default values. Figure 1 also contains for each of the viewtypes an illustrative example view showing a simple model. The figure shows how viewtypes relate to classes (BusinesObject) and views to instances of these classes (Customer, Address). The language's dynamic viewpoint has an interaction viewtype that defines how to represent business object interrelations and a block implementation viewtype that defines how behaviour of single business entities is specified. Again, Figure 1 illustrates each viewtype on an example.

Using this example, we introduce our terminology illustrated in Figure 2. The class digram shows the terms used in view-based modelling and represents our tool-centric understanding of views, viewtypes, and viewpoints.

Our conceptualisation starts with the *System* (or the "real-world object") which is being studied by its *Stakeholders* wrt. their specific system *Concerns*. In our example, we may think of database designers of the system under study who want to analyse their database table structure. Therefore, they are interested in the static *Viewpoint*. A viewpoint represents a conceptual perspective that is used to address a certain concern. A view point includes the concern, as well as a defined methodology on how the concern is treated, e.g., instructions how to create a model from the particular viewpoint. In order to analyse the system, they create a single, consistent *Model* of the system under study. The model has to be an instance of its *Metamodel*.

In order to show parts of this model, we need a set of concrete syntaxes. These concrete syntaxes are defined by *Viewtypes*. It defines the set of metaclasses whose instances a view can display. This description uses the metamodel elements and provides rules that comprise a definition of a concrete syntax and its mapping to the abstract syntax. It defines how elements from the concrete syntax are mapped to the metamodel and vice versa. For example, the business object structure viewtype shows classes and their relations but not the instructions how to persist the entity to the database.

A *View* is the actual set of objects and their relations displayed using a certain representation and layout. A view resembles the application of a view type on the system's models. A view can therefore be considered an instance of a view type. For example, the structure view in Figure 1 shows the business entities "Customer" and "Address". They may be a selection of all possible classes, e.g., "CreditCardData" is not shown on this particular view but may be shown on a different view from the same view type. Also the elements "Customer" and "Address" can also appear in other views.

The separation between the definition of the view type and its instances is also topic of the recently started initiative of the OMG called Diagram Definition [Obj10]. Within this new standard, which currently under development, the OMG distinguishes between Diagram Interchange (DI) and Diagram Graphics (DG). Where the former is related to the information a modeller has control over, such as position of nodes, the latter is a definition of how the shapes of the graphical language look like. The mapping between a DG and a metamodel can then be defined by a mapping language such as QVT. Considering a DI instance a view and a DG definition including the mapping a view type the OMG's definition is perfectly in line with our own experience and what we contribute in this paper.

The ISO 42010:2011 standard [ISO11] gives the following definitions: A *view* "adresses one or more of the concerns of the system's stakeholders" and "expresses the architecture of the system-of-interest in accordance with an architecture viewpoint" where a *viewpoint* "establishes the conventions for constructing, interpreting and analyzing the view to address concerns framed by that viewpoint." In contrast to the ISO 42010:2011 standard [ISO11], we follow the idea of having a *single* model of the system under study and views just *visualise and update* this central model. For different kinds of data, we favour the use of *different* view types, in analogy to the model type in the ISO standard. Furthermore, we do not focus on architecture descriptions. As a consequence, our concept contains explicit the model and its metamodel which also allows us to associate the viewtypes to the metamodel. As a minor difference, we favour a view point just to address a single concern to have a clear relation.

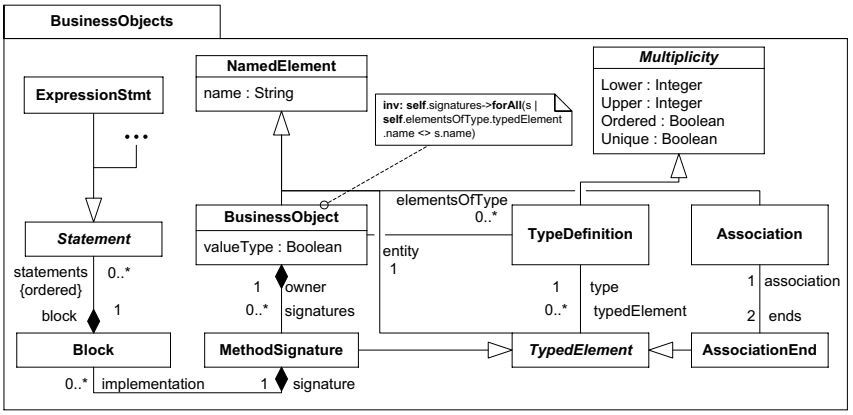


Figure 3: Example metamodel.

3 Classification of View Type, View and Editor Properties

We explicitly distinguish between the definition level (view type) and the instance level (view) to pay respect to the different roles involved in domain specific modelling. Modellers use views on instance level, whereas language engineers work on view type level. To separate between the properties of these levels we first present view type properties in Section 3.1 and second view properties in Section 3.2. Finally, as DSML frameworks mostly come with their own editor framework, which also has a large impact on the way how modellers can work with their views, we present a classification scheme for editor capabilities that have an impact on view building in Section 3.3.

We use the properties presented here for two different purposes. Firstly, for the communication and reasoning about specific view types and views. Having these explicit properties eases the communication and helps to avoid errors in the definition as well as the application of a view-based modelling approach. Secondly, applied to a given view-based modelling approach, the fulfilment of a property resembles the fact that a certain approach is capable of defining view types or instantiating views that feature this specific property.

3.1 View Type

A view type defines rules according to which views of the respective type are created. These rules, can be considered as a combination of projectional and selectional predicates as well as additional formatting rules that determine the representation of the objects within the view. Projectional predicates define which parts of a view type's referenced metamodel and/or elements of that metamodel a view actually shows. For example, the "BusinessObject Structure" view type defined in our example is a projection that shows elements of type `BusinessObject`, `Association`, etc. but not `Block` or `Statement` elements. Additionally, projectional predicates may also refer to specific attributes defined on metamodel level. In other words, projectional predicates define which types of elements (classes, associations, attributes, etc.) a view type includes.

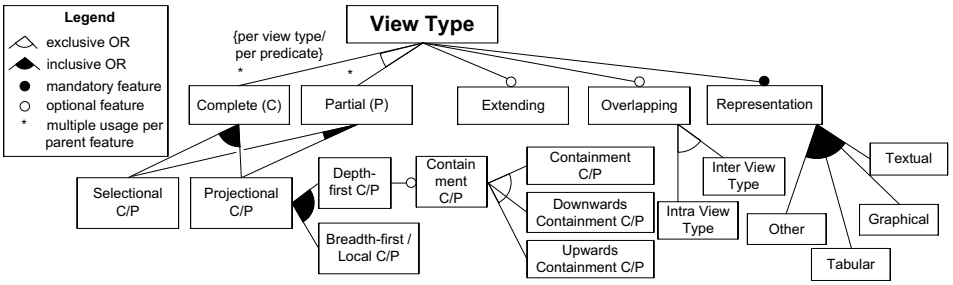


Figure 4: Properties of view types.

Selectional predicates define filter criteria based on attributes and relations of a view's elements. For example, the "ValueTypes Overview" view type of our example only includes elements of type `BusinessObject` that have set their `valueType` attribute to `true`, thus showing only abstract classes. In other words, selectional predicates define on instance level, which conditions elements have to fulfil in order to be relevant for a specific viewtype.

Finally, a view type contains rules defining how a view represents the projected and selected elements. Given the "BusinessObject Structure" view type of our example, one of these rules describes that the view type displays `BusinessObjects` as rectangular boxes with the value of their `name` property as label.

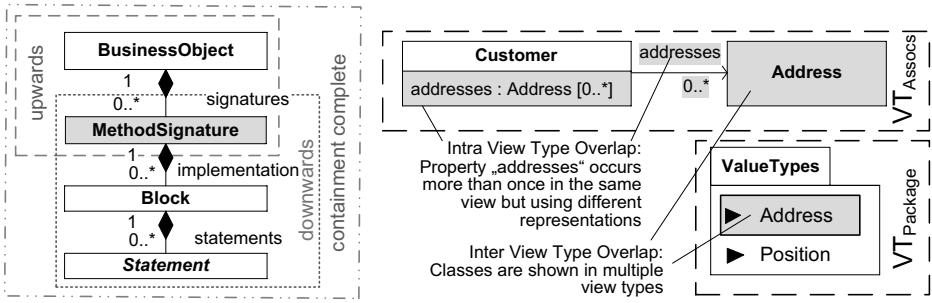
The determination of the three rule types is given on a conceptional level, the implementation of a view type may also combine these rules into a joined rule. These three types of rules define the range of properties a view type may fulfil. Examples are projectional or selectional completeness or whether a view type defines a textual or graphical representation of the underlying model.

Note that, depending on whether a view should be editable or read-only, these rules have to be considered in a bidirectional way: I) the direction which specifies how a view is created for an underlying model II) defines how a model is created and/or updated based on changes that are performed in a view.

The feature diagram depicted in Figure 4 gives an overview of the properties identified by us.

Complete View Type Scopes: A language engineer needs to ensure that the created language is capable of expressing programs of the targeted domain. Especially if a view-based approach is employed, achieving the desired expressiveness and coverage of the domain's metamodel can be an error prone task. To ease the communication on this coverage as well as to provide a basis for tool builders to cope with this challenge, we define different types of completeness for the definition of a view type.

A view type may be *complete*, which means that it considers all classes, properties, and relations of a metamodel that are reachable from the part of the metamodel for which the view type is defined. A *complete* view type can be used as a starting point for the interaction of a model displaying all model elements from which a modeller can dive deeper into the model using other view types.



(a) Example: containment complete.

(b) Examples: view type overlaps.

Figure 5: Examples for containment completeness and view type overlaps.

The scope of a view type rule is given by its projectional as well as selectional parts. Thus, we can also distinguish between *projectional* completeness (which includes the *containment* and *local completeness*) and *selectional* completeness (see the definition of *instance completeness* below).

Projectional Complete View Type Scope: Projectional completeness of views based on the MOF [Obj06] meta-metamodel can be defined on several levels. Starting from a certain model element there are two dimensions on how to span a scope to other model elements. First, traversing *depth-first* to other model elements via the associations defined in the metamodel and second, including, all elements that are, *breadth-first*, reachable through all directly attached attributes or associations.

Depth-first completeness is hard to specify as it would need to be defined for each specific path through a metamodel. However, a special subset of depth-first that is useful to define is the completeness w.r.t. to the `containment associations`. Containment associations are the primary way for creating an organisational structure for a metamodel. Therefore, we define a special case of depth-first completeness called *containment-completeness*. A view type is containment complete concerning a specific element o if all elements that are related to it via containment associations are shown in the view. Three different notions of containment complete can be defined (Figure 5(a) shows examples based on our example metamodel.).

- Downwards containment-complete means that all elements that are transitively connected to o are part of the view if o is their transitive parent.
- Upwards containment-complete means that all elements that are transitively connected to o are part of the view if they are a transitive parent of o.
- The third notion specifies that all transitive parents and children of o are part of the view.

The second dimension of view type completeness is the *breadth-first* completeness, which we call *local completeness*. Local completeness is fulfilled if a view type can display all directly referenced elements of a given element. A view type is locally complete concerning a class c if every direct property of c can be displayed by an instance of the view

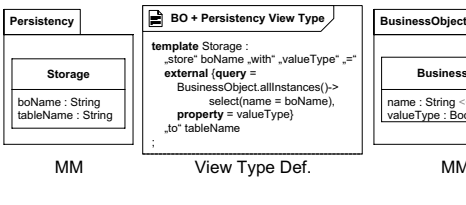


Figure 6: Using an external metamodel *Persistency* that is connected via a query in the view type to an existing metamodel *BusinessObjects*.

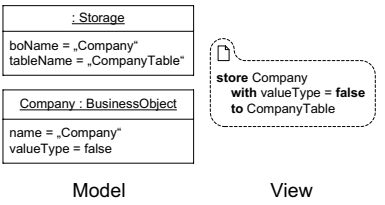


Figure 7: Example view instance of the view type specified in Figure 6.

type. Making this type of completeness explicit is useful if a certain view type should be a detail editor for a specific class where the modeller should be able to view and/or edit all properties of the given class.

Selectional or Instance Completeness: Selectional completeness, or *instance* completeness means that the selection of the view type includes all model instances that appear in the underlying model as long as the projection of the view type also includes them. However, *projectional completess* is not required in order to fulfil the *instance* completeness property. For example, a view type can have a projection of a class A which does not include a property $propA$. As long as the view type includes all possible instances of A it is still *instance complete*. In contrast to that, if a view type defines a selection criterion for A, such that only As having a $propA$ value of "selected", are included the template for A is not *instance complete* anymore.

Partial View Type Scope: The scope of a view type is considered *partial* concerning a metamodel, if it only covers a certain part of the element types that are defined within the metamodel. This means, for example, the "BusinessObject Structure" view type of our example is *partial* w.r.t. the metamodel as it omits the classes such as `Block`. Just as a view type has different types of completeness, i.e., projectional and selectional completeness, a view type that is not complete w.r.t. one of these properties it is then automatically *partial*.

Extending View Type Scope: In addition to the properties *partial* and *complete*, there are also view types that combine elements from the underlying model with additional information from an external model M_{ex}. The extended information is defined by the fact that it is not directly reachable by model navigation but by some kind of external model, e.g., a decorator model, from the extended view type. Often, the information that should be added in such a view type is additionally defined using a different metamodel MM_{ex}. In our terminology a view type always refers to a single metamodel. Therefore, the metamodel for such an *extending* view type refers to an artificial composite metamodel including both related metamodels.

A concrete example that shows how such an extension view type could be defined is depicted in Figure 6, based on the example business object metamodel. The example shows that there is an external *persistency* annotation metamodel that does not have any connection with the business object metamodel. The storage annotation only contains a hint to the name of the business object that should be persisted in its *boName* attribute. However, it might be a requirement that a language engineer needs to define a view type not only showing elements of the persistency metamodel but also presenting information from the

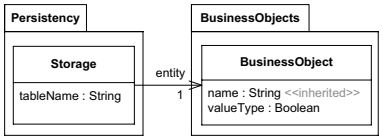


Figure 8: Using an external metamodel *Persistency* to non-intrusively add persistency annotations to an existing metamodel *BusinessObjects*.

business object metamodel, i.e., if the mentioned business object is a value type or not. Therefore, a query is given in the view type that retrieves the corresponding business object with the specified name and from which the *valueType* property is then shown in the view, as illustrated in Figure 7.

Overlapping View Type Scope: This property is not a direct property of a view type but defines a relationship between two or more view types. View types may also cover scenarios where there is more than one view type that is able to represent the same type of element. On the other hand it is also possible that the same view type can handle a distinct type of element in different ways. We call these types of overlaps *inter-* and *intra view type overlap*. Figure 5(b) shows an examples for both types of overlaps.

An *inter view type overlap* occurs whenever one or more view types are able to represent the same element. A prerequisite for this property is, that the involved view types are based on the same metamodel. Figure 5(b) shows that the "BusinessObject structure" view type has an inter view type overlap with the "ValueTypes Overview" view type as both show `BusinessObjects`.

If the same view type can represent the same element in different ways an *intra view type overlap* is present. This means that there is more that one predicate in the view type that includes the same element. Figure 5(b) shows that the "BusinessObject structure" view type has an intra view type overlap as association ends are represented as a compartment within the `BusinessObject's` shape as well as within a label decoration of the `Association's` shape.

Representation: The third type of rules which a view type defines are responsible for defining the representation of the elements of a view. A view may comprise different types of representation rules. Possible types are *textual*, *graphical*, *tabular* and arbitrary *other* types. Rules may also combine different types. For example, a graphical representation may include some textual or tabular parts as well.

3.2 Views

Views, as instances of view types can also have different properties which are depicted in Figure 9 using a feature diagram. Using these properties, we classify views concerning the extent of information they show of the underlying model(s). We distinguish between selective and holistic views. Additionally, we handle the persistence and editability of layout and selection as well as inter and intra view overlaps of views. The following

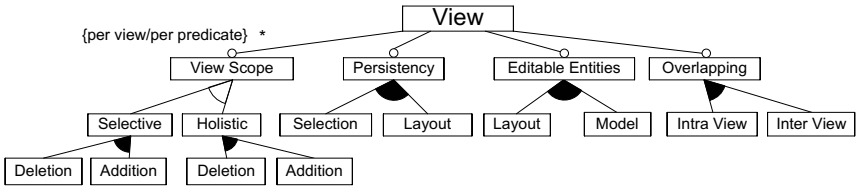


Figure 9: Properties of view instances.

properties affect the behaviour of the a view *instance*, however, also the view type may define generically whether a view has a specific property or not.

View Scope: A view shows a specific selection of elements from its underlying model. If changes occur in the model, i.e. elements are added or deleted, the view needs to be updated according to these changes. The *view scope* property defines whether this is done automatically or only if a user explicitly requests the update.

Selective View Scope: A view is considered *selective* if it is possible to show a subset of the elements that could be shown according to its view type. A selective view only shows these specifically selected elements. The selection may either be done automatically or manually by a user of the view. For example, the view example for the "BusinessObject structure" view type depicted in Figure 1 is selective, as the modeller can manually select whether or not specific BO occurs in the view or not. In this case the view only shows the BOs "Customer" and "Address" and omits, for example, "Company".

A view can be selective concerning different types of changes:

Addition Selective Addition of elements to the model that fall into the scope of a view's view type are only added to the view's selection, if added manually. For example, the "BusinessObject Structure" view types may not show all BOs at once. A modeller can select whether a newly added BO should appear in a certain view or not.

Deletion Selective Deleting of elements from the model that fall into the scope of a view's view type are propagated to the view's selection, if deleted manually. Thus, elements that were deleted from the underlying model do not necessarily result in the deletion of their view representations. For example, in many graphical modelling tools, representations of elements in a view where the underlying model element is not available anymore, are not automatically removed from the diagrams but are rather annotated, indicating that the underlying model element is missing.

Holistic View Scope: In contrast to addition selective views, a view may be *addition holistic*. This means that it always presents the whole set of possible elements that can be displayed by the view. If elements are added and/or removed this is immediately reflected automatically in the view. Modelling tools mostly use this type of view to present the user an overview on the underlying model. Analogously, *deletion holistic* views directly reflect any deletion of an element by removing its view representation, i.e., view and model are always synced.

Overlapping: This property is not a direct property of a view but defines a relationship between two or more views. A view may be *overlapping* with another view. In this case

elements may occur in more than one view at once. This may be a view of the same view type but also a different one. If the element occurs in multiple views we speak of *inter view overlap*, whereas we call multiple occurrences within the same view *intra view overlap*.

Editability: In addition to displaying model elements according to the view type's rules, an *editable* view needs to provide means to interact with and thus modify the underlying model. Actions such as create, update and delete need to be performable to make a view editable. Editability of views can also be subdivided into two different degrees of editability. First, if only the layout information can be changed but not the actual model content, the view is only *layout editable*. Second, if the model content is editable through the view it is considered *content editable*.

Another interesting aspect is that editability is closedly related to the view type scope (cf. Section 3.1). The scoping of a view type might dertermine the editability of its views. For example, a view type might omit a mandatory attribute of a metamodel class in its specification. In this case it is not possible to create new instances of this class using this view type but it is still possible to view and modify instances of the class.

Persistency: A view may be *persistent* regarding its *selection* as well as its *layout*. Stored view layouts enable faster access, as it does not need to be created newly every time a modeller opens the view. Additionally, if a *persistent* view it is at the same time *editable* enables for customisation of a view's selection of elements and/or layout.

For non-holistic views, the modeller decides which elements a view should include and which not. If such a selection should be saved, a view needs to be *selection persistent*. In this case the view's selection of elements is stored.

Additionally, a modeller may customize the layout of the view by manually changing certain parts, such as explicit positioning of the elements occurring in the view, or, for textual views, white-spaces or indentations. Additionally, a modeller may add additional, mostly informal content, such as comments or annotations. If a view allows to store this kind of information it is *layout persistent*.

3.3 Editor Capabilities

Features that have an impact on how view-based DSML frameworks deal with the interaction of users and views as well the synchronisation between view and model also influence the requirements on an employed view-based modelling approach. Figure 10 depicts a feature diagram that gives an overview on these *editor capabilities*. Note that we included only such properties we consider as special requirements for a view-based modelling approach. A broader view on DSML editor capabilities, at least for textual DSML approaches can be found in [GBU08].

Bidirectionality: To keep models and their views in sync, the rules that do this synchronisation need to be bidirectional (or there need to be two rules where one resembles the inversion of the other). In order to be correct, a bidirectional rule (or a pair of corresponding rules) needs to comply to the *effect conformity* property. To comply to effect conformity, changes made directly to the model should leave it in the same state as an equal change on the view level that is then automatically propagated back to the model would do. This

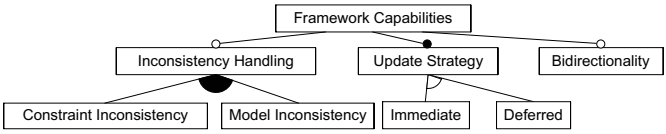


Figure 10: Editor capabilities.

automated back propagation is defined by the view type rules. Furthermore, vice-versa, changes made through the view to the model should leave the view in the same state as an equal change on the model level which is propagated by the corresponding view type rule. Additionally, Matsuda et al. [MHN+07] define three *bidirectional properties* that need to be fulfilled in order to create consistent view definitions.

Update Strategy: We classify the update strategy which triggers the propagation of changes between view and model into two different types. (I) An update can be performed at the very moment a change is made to one of the sides, either model or view. This kind of update is denoted an *immediate* update strategy. (II) An update may occur at a point in time decoupled from the actual change event. This kind of update is denoted *deferred* update strategy.

The point in time when updates are performed predefines the number of allowed changes between two subsequent synchronisation runs. In the *immediate* update strategy the transformations are executed as soon as an atomic change was performed to either the view or its model. This strategy allows a tighter coupling between view and model and avoids conflicts that may occur if an arbitrary number of changes is performed before the next synchronisation. On the other hand, the *deferred* update strategy allows to have an arbitrary number of changes in this time span. This allows to work with views in a more flexible way, as they can be changed offline, i.e., if the underlying model is currently not available. However, having an arbitrarily large number of changes, that need to be synchronised, dramatically increases the probability of conflicts.

Consistency Conservation: As modelling is a creative process, models are mostly created step-by-step. Thus, allowing for intermediate, possibly inconsistent states may foster the usability and productivity of a view-based DSML [Fow05, FGH+94]. If this is the case, an editable view might contain valuable information that was created during modelling but that is not yet transformable into a valid model.

We define two different classes of inconsistency: (I) violation of metamodel constraints that lead to what we call *constraint inconsistency*, which means that the view has statically detectable semantic errors and (II) *model inconsistency* if a view is syntactically incorrect and cannot be transformed into a model at all.

Metamodel constraints restrict the validity of models that would theoretically be constructible obeying only the rules defined in the metamodel without constraints. This also includes multiplicity definitions for associations and attributes. For example, considering out example metamodel, an invariant defined for the metamodel class `BusinessObject` expresses that a `MethodSignature` may not have the same name as an `AssociationEnd` connected to the same BO (expressed as OCL invariant: `inv: self.signatures ->forAll(s | self.elementsOfType.typedElement.name <> s.name`

)). If there exists an instance of `MethodSignature` which has a name that is already given to such an `AssociationEnd`, this constraint is violated. However, during the process of modelling there may be intermediate states where both elements have the same name, e.g., during a renaming process. Still, the element should be representable in a view, i.e., with additional information stating that the constraint is currently violated. If constraint inconsistency was not supported, the modeller would have to first change the `AssociationEnd's` name before renaming the `MethodSignature`.

In case (II) a greater degree of freedom in modelling can be reached if a view even supports to hold content that cannot be translated into a model at all. This allows a developer to work with the view like a "scratch pad". We denote this type of inconsistency *model inconsistency*. As graphical modelling tools mostly only allow the modeller to perform atomic modifications that preserve the syntactical correctness of the view, this type of inconsistency mostly only occurs within textual modelling. In the latter case modellers are often free to type syntactically incorrect information within a view.

4 Related Work

Oliveira er al. [OPHdC09] presented a theoretical survey on DSLs which also included the distinction between the language usage and development perspectives. However, the presented survey remains on a more conceptual level, mostly dealing with properties such as internal vs. external, compilation vs. interpretation as well as general advantages and disadvantages of DSL approaches. The authors do neither mention graphical DSMLs nor do they include view-based modelling aspects in their survey.

Pfeiffer and Pichler give a tool oriented overview on textual DSMLs in [Pfe08]. Their survey is based on three main categories, which are language, transformation, and tool. The evaluated features include the representation and composability of language definitions, transformation properties such as the update strategy as well as the kind of consistency checking that is supported. However, view-based modelling aspects and graphical or hybrid DSML approaches are omitted.

Buckl et al. [BKS10] have refined the ISO 42010 standard and created a framework for architectural descriptions. A formal definition of the terms *view*, *viewpoint* and *concern* is provided, which is in compliance with ISO 42010. The definition is however restricted to architecture modeling.

In our own previous work [GBU08] we conducted a classification based survey on textual concrete syntax approaches which have a common intersection with the approaches for view-based DSMLs. However, the focus in this previous work was on evaluating the textual modelling capabilities such as grammar classes, generator and editor capabilities (such as code completion or syntax highlighting) and did not include features that are required for view-based modelling.

Another, feature based survey on textual modelling tools, is presented by Merkle in [Mer10]. Although, this survey includes some features, which we also discuss, such as the representation of the concrete syntax as well as some tool related aspects, it does not present view related features nor does it give hints on the existence of view-based aspects in the

classified tools.

5 Conclusions & Future Work

In this paper, we identified properties for the main concepts of view-based DSMLs. The analysis was based on our experiences with several different graphical, as well as textual DSML approaches. In this we distinguish between viewpoints, view types and views. We furthermore focus on properties that relate to tool oriented capabilities such as partial or overlapping view definitions or holistic and selective views.

This classification scheme allows DSML developers and users to explicitly specify properties of view types and views. This enhances the communication between language engineers and modellers during requirements elicitation, specification and implementation of view-based DSMLs.

Based on the classification scheme we will carry out a systematic review of existing DSML approaches. The results of this analysis are beneficial for language engineers as it helps them in selection process of a view-based modelling approach. Furthermore, we will be able to identify gaps in tool support w.r.t. view-based modelling. Preliminary results of the tool evaluation are available online.[2]

References

[ASB09] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Supporting View-Based Development through Orthographic Software Modeling. In Stefan Jablonski and Leszek A. Maciaszek, editors, *ENASE*, pages 71–86. INSTICC Press, 2009.

[BKS10] Sabine Buckl, Sascha Krell, and Christian M. Schweda. A Formal Approach to Architectural Descriptions – Refining the ISO Standard 42010. In *Advances in Enterprise Engineering IV*, volume 49 of *Lecture Notes in Business Information Processing*, pages 77–91. Springer Berlin Heidelberg, 2010.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, first edition, 2007.

[Cle03] Paul Clements. *Documenting software architectures: Views and beyond*. SEI series in software engineering. Addison-Wesley, Boston, Mass., 2003.

[Ecl11a] Eclipse Foundation. Graphical Modeling Framework Homepage. `http://www.eclipse.org/gmf/`, 2011. Last retrieved 2011-10-06.

[Ecl11b] Eclipse Foundation. Xtext Homepage. `http://www.eclipse.org/Xtext/`, 2011. Last retrieved 2011-10-06.

[FGH+94] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling In Multi-Perspective Specifications. *IEEE Transactions on Softw. Eng.*, 20:569–578, 1994.

[2] http://sdqweb.ipd.kit.edu/burger/mod2012/

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2, 1992.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? 2005.

[GBU08] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications*, pages 169–184, 2008.

[GHZL06] John C. Grundy, John G. Hosking, Nianping Zhu, and Na Liu. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In *ASE*, pages 25–36. IEEE Computer Society, 2006.

[ISO11] *ISO/IEC/IEEE Std 42010:2011 – Systems and software engineering – Architecture description*. Los Alamitos,CA: IEEE, 2011.

[KT08] S. Kelly and J-P. Tolvanen. *Domain-Specific Modeling:Enabling Full Code Generation*. Wiley-IEEE Society Press, 2008.

[KV10] Lennart Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proceedings of OOPSLA*, pages 444–463, 2010.

[MCF03] S.J. Mellor, A.N. Clark, and T. Futagami. Model-driven development - Guest editor's introduction. *IEEE Software*, 20:14– 18, 2003.

[Mer10] Bernhard Merkle. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of SPLASH*, pages 139–148, New York, NY, USA, 2010. ACM.

[MHN+07] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectional Transformation based on Automatic Derivation of View Complement Functions. In *Proc. of the ICFP 2007*, page 47//58. ACM Press, 2007.

[MPS] JetBrains MPS. `http://www.jetbrains.net/confluence/display/MPS/Welcome+to+JetBrains+MPS+Early+Access+Program`.

[Obj06] Object Management Group (OMG). MOF 2.0 Core Specification, 2006.

[Obj10] Object Management Group (OMG). Diagram Definition, 2010.

[OPHdC09] Nuno Oliveira, Maria Joao Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Domain Specific Languages: A Theoretical Survey. In *Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, 2009.

[Pfe08] A Comparison of Tool Support for Textual Domain-Specific Languages. In *8th OOPSLA Workshop on Domain Specific Modeling*, 2008.

[RW05] Nick Rozanski and Eoin Woods. *Software Systems Architecture*. Addison-Wesley, 2005.

[Szy02] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., 2002.