An ADT Profile

Marijn Temmerman[(1)], Serge Demeyer[(2)], Francky Catthoor[(3)]

(1) Karel de Grote-Hogeschool, Salesianenlaan 30 Hoboken, Belgium
(2) Dept. of Mathematics and Computer Science, University of Antwerp, Belgium
(3) IMEC vzw, Kapeldreef 75 Heverlee, Belgium

This report presents our ADT profile, a UML profile for modeling the data-structure related aspects of an application. The main goal is to support global data-structure optimization activities at the modeling level.

We implemented this ADT profile using MagicDraw UML 12.1 [15], a visual UML 2 modeling and CASE tool that is widely adopted in the software industry. MagicDraw supports the XML Metadata Interchange (XMI) 2.1 standard for the distribution of UML profiles and UML user models. This ADT profile is available at the website of the LORE research group in XML format for import and ready for use in other UML 2 compliant modeling tools [16].

This report is organized as follows: Section 1 gives an overview of the structure of the ADT profile. In the following three sections, the new ADT-specific modeling concepts are discussed in detail and illustrated with clarifying examples.

1 Structure of the ADT Profile

We designed this ADT profile according to the most recent specification of the UML language [1].

Figure 1 shows the structure of our ADT profile. We organized the elements of the ADT profile into three subpackages, namely one model library and two profiles, that each deal with specific ADT-modeling aspects. A number of primitive data types are collected in the model library named ADT Primitive Types. ADT Classes is the profile that contains the stereotypes that are to be used in user-defined ADT class diagrams. Finally, to annotate the dynamic properties of the data structures in the sequence diagrams, the ADT Behavior profile is supplied.

The ADT profile references the UML metamodel subpackages Models, UseCases, Classes, and Interactions. Hence, when a user ADT model (i.e. a user-defined package) applies this ADT profile, all the standard UML model elements necessary for constructing use-case diagrams, class diagrams, and sequence diagrams are available (see also Fig. 3). The unreferenced standard UML model elements are hidden when the ADT profile is applied and can thus not be used in a user-defined ADT model.

Below, each of the subpackages of the ADT profile is discussed in detail. The descriptions of the individual stereotypes are broken down into parts corresponding to different aspects. In cases where a given aspect does not apply, its part may be omitted entirely from the description. The following parts are used to specify a stereotype:

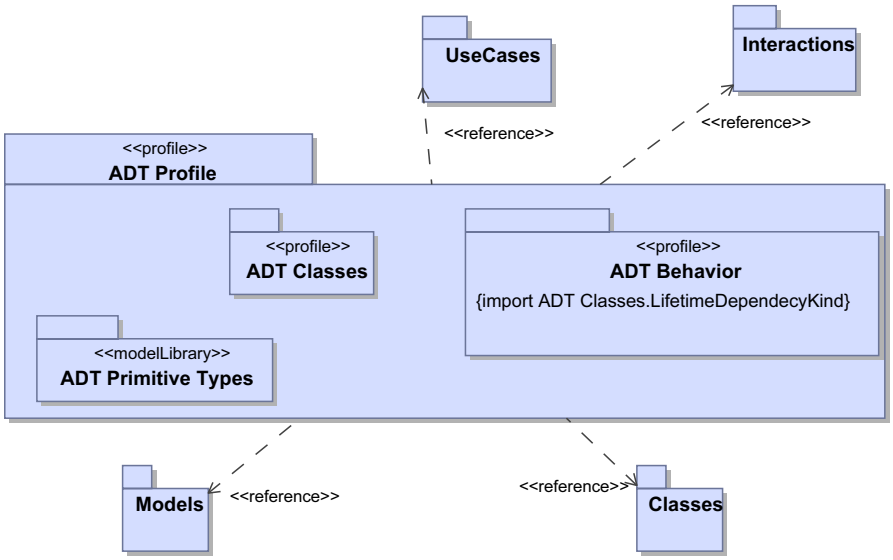

Figure 1: Structure of the ADT profile.

- The heading gives the formal *name* of the stereotype or concept.
- A description of the meaning of the new concept.
- The *Tag* part description lists each of the meta-attributes that are defined for that stereotype. Each tag is specified by its formal name, its type, and multiplicity. If no multiplicity is listed, it defaults to 0..*. This is followed by a textual description of the purpose and meaning of the tag. If a tag is derived, the name will be preceded by a slash.
- The *Constraints* part contains a list of all the constraints that define additional well-formedness rules that apply to this concept. Each constraint consists of a textual description.
- The *Notation* part gives the basic notational forms used to represent the concept and its features in diagrams. Only concepts that can appear in diagrams will have a notation specified. This typically includes a simple example illustrating the basic notation.
- The *Example* subsection, if present, includes additional illustrations of the application of the concept and its notation.

2 The ADT Classes Profile

The ADT Classes profile collects all the stereotypes that deal with the structural view of a user-defined ADT model, in particular ADT-level class diagrams, ADT classes and their relations. Figure 2 gives an overview of the contents of this package. The UML metaclasses that are extended by the stereotypes are indicated between square brackets for compactness.

The stereotypes Clientship, Concept, and Container serve for modeling the ADT-level structure of an application. The interface of the containers can be precisely specified by means of the stereotypes add, remove, select, isEmpty, and foreach. A composite ADT operation can be concisely formulated with the COP stereotype. To select an element from a container a Key can be specified. We provide the Configuration stereotype to indicate a particular configuration of the elements of a container, when adding and removing an element from a container.
Below, each of these stereotypes is discussed in detail. They are arranged in an order that enables sequential reading.

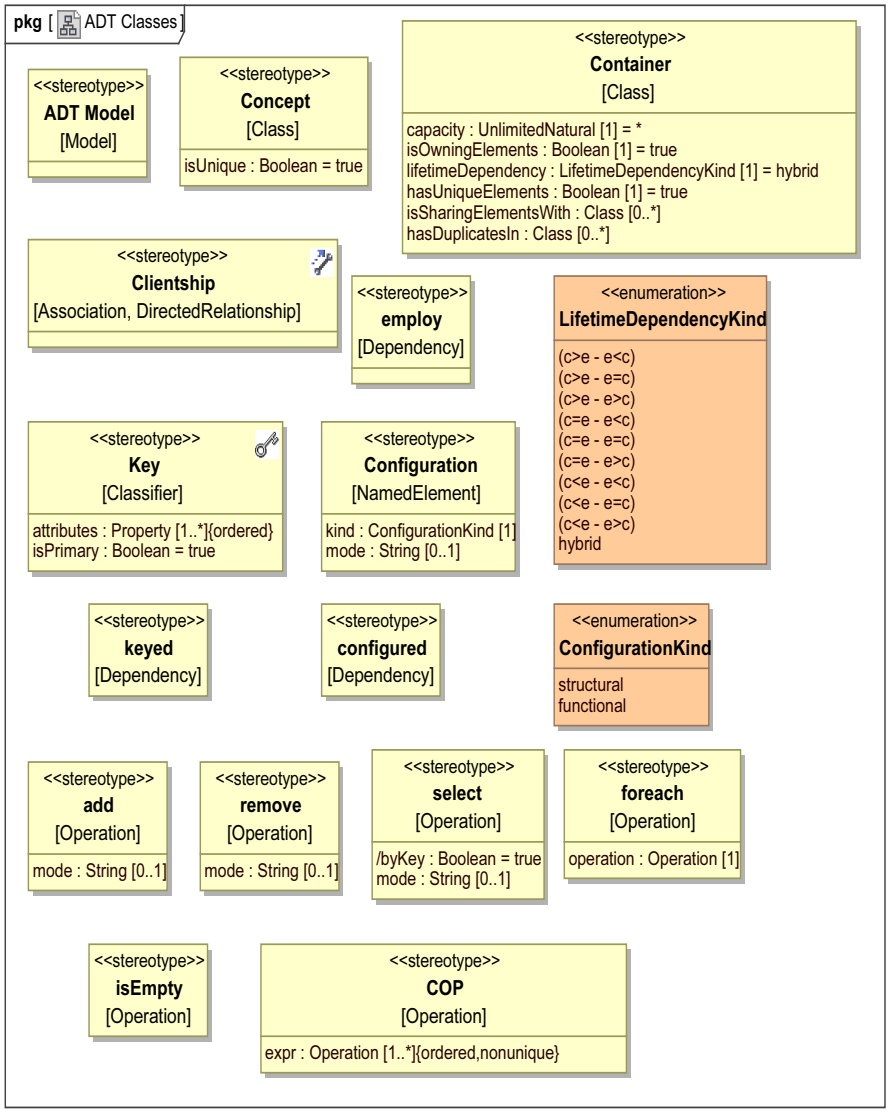

Figure 2: Contents of the ADT Classes profile.

2.1 ADT Model

The stereotype ADT Model is a special kind of Model. A user-defined ADT model (i.e. an instance of the stereotype ADT Model) is a high-level representation of a system that focuses on the data-structure related aspects of the system and this for a specific and dominant runtime situation. From now on we will call this runtime situation a *scenario.*

An ADT model covers the following three aspects of the system :

1. the specific scenario of the system that is subject of the optimization,
2. a model of the data structure of the system, expressed in terms of user concepts and collections of objects (i.e. the containers) that are involved in (1),
3. a high-level description of the interaction of the collaborating objects during (1).

Therefore, an ADT model contains three different types of UML diagrams:

1. a use-case diagram that specifies the scenario (e.g. Fig. 3b),
2. a class diagram that presents the ADT-level data-structure model of the system,
3. a sequence diagram that depicts the sequence of the high-level data-access patterns of the objects and containers for the scenario.

The purpose of an ADT model is to represent (in a precise way) a global data-structure model of an application (or part of). The ADT class diagram illustrates the following two aspects: (i) it shows how the data of the application is classified into related concepts and containers, and (ii) it shows which operations (i.e. the interface) are required by the concepts and the containers with respect to the considered usage of the system. The ADT sequence diagram describes how the collaborating objects and containers realize the functionality of the scenario that is specified in the use case diagram.

Tags

- no tags

Constraints

- A user-defined ADT model must apply the ADT profile.

Example

Figure 3a shows the ADT model for a Mesh rendering application, which is a user-defined model package. `Mesh` is thus an instance of the stereotype ADT Model, which is indicated by <<ADT Model>>. By applying the ADT profile, indicated by the <<apply>> relationship, all the ADT-specific modeling elements become available for use within the ADT model `Mesh`.

The `display` scenario of `Mesh` is specified by means of a standard UML use-case diagram (see Fig. 3b).

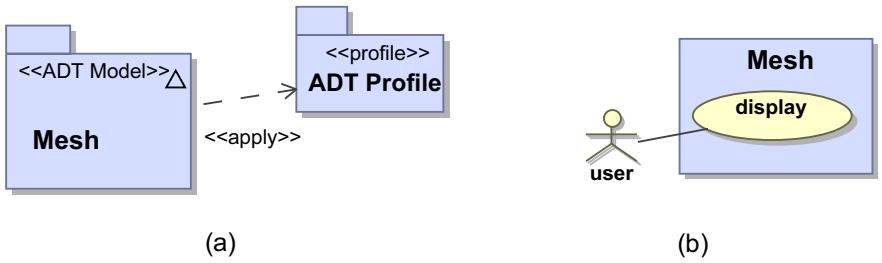

Figure 3: (a) The ADT model `Mesh` applying the ADT profile, and (b) specifying the `display` scenario of `Mesh` in a use-case diagram.

2.2 Clientship

UML associations can vary in many ways, but at the concrete data-type level (i.e. the source-code level) there are only a few variations. Bunse expressed it as follows in [2]: ”All inter-object relationships are essentially implemented by the same basic mechanism: one object holding a pointer to, or the value of, another object”.

Inspired by Bunse’s NOF[1] metamodel [3], this ADT profile offers just one kind of association between classes: the *Clientship*. A clientship is a directed, binary association between a Client class (the source) and a Server class (the target): the client is aware of the existence of the server, but the server has no knowledge of the client.

In an ADT class diagram, the UML associations (i.e. only the navigable association ends, because these will be implemented) of the conceptual model are refined into clientships. If the UML association end has a multiplicity range larger than one, then also a *Container* class is introduced (see Section 2.3). In essence, a clientship (together with a container) is used to eliminate the multi-valued multiplicities of the conceptual model.

However, a clientship does not imply any source-code level decision. An ADT clientship only indicates the existence of a compilation-time analyzable (and thus static) structural relationship between two classes. The clientship gives the client class access to the server. The server has no access to the client.

Tags

- no tags

Constraints

- Every association in a valid ADT class diagram must be a clientship.
- The client class of a clientship relation must be a Concept.
- The server class must be a Concept or a Container.
- The upper bound of the multiplicity range at the server side end of a clientship must be 1.

[1]The NOF is aimed at refining structural UML models into more concrete models with the same level of detail as in the (object-oriented) source code.

Notation

In an ADT class diagram, a clientship (Fig. 4) is shown using a black filled arrow pointing to the server class. The use of the keyword < <Clientship> > is optional.

Figure 4: A clientship relation gives the Client access to the Server class.

Example

Figure 5a shows a part of a conceptual model that is taken from the Mesh example. The class `Mesh` is directly related to the class `Triangle` by means of a UML association. The many multiplicity range (i.e. '*') at the `Triangle` side of the association indicates that multiple triangles are used to compose a mesh.

In the ADT-level class diagram (Fig. 5b), the concept `Mesh` (the client) is related by a clientship to the container `Faces` (the server). `Mesh` is an instance of the stereotype Concept (see Section 2.3), which is indicated by < <Concept> >. `Faces` is an instance of the stereotype Container (see Section 2.4). `Faces` is the class that collects the `Triangle` objects and is the abstraction of the data structure type that is required at the source-code level. The clientship gives the mesh access to the container object that manages the triangles.

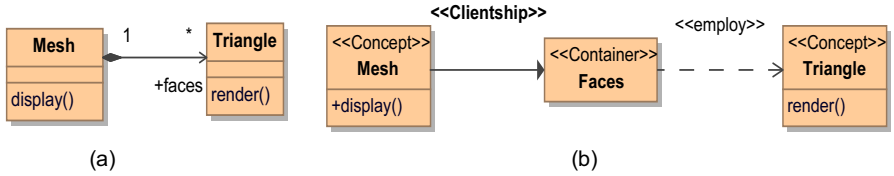

Figure 5: (a) An association in the conceptual model of Mesh that relates the classes `Mesh` and `Triangle`, and (b) the refinement of this association in the ADT model of `Mesh` by means of a clientship and the container `Faces.`

2.3 Concept

A Concept is special kind of Class. The UML specification states that an instance of a class embeds the actual state or values of its attributes [1]. This implies that an attribute contributes to the memory footprint of a runtime instance of the concept, unless it is implemented in an implicit way, e.g., as the index of an array.

Concepts are the models of the different kinds of atomic data elements – also known as data records [4, 5] – with which the runtime data structure of the system is composed.

For a valid Concept, to enable the calculation of the memory-related cost metrics of the data objects, the type of an attribute is restricted to the ADT primitive types and the enumeration types (see Section 3).

Multivalued attributes, which represent collections of values (like multivalued association ends do), are not allowed and must be refined by means of Containers. In this way, their implementation variations can be explored at the concrete data-type refinement stage.

Attributes of concepts may be derived. They are used to indicate that their value can be derived from other data in the model. A derived attribute is shown by placing a slash ('/') in front of the name of the attribute. Attributes that are not indicated as derived are assumed to be essential.

Because encapsulation of the data is not a primary point of interest for ADT modeling, we decided to fix the visibility of the attributes to the value 'public'. As a consequence, it is rather useless to show the visibility of the attributes and operations in ADT class diagrams.

The ADT profile does not consider inheritance. Inheritance is a fundamental concept in object-oriented modeling and programming, but its application is not 'visible' anymore when the data is stored in the memory. Therefore, concepts must not be extended. Existing inheritance relationships in the conceptual model of the application must be first resolved by means of associations. We refer to the catalogs of Blaha [6] and Lano [7] that present various behavior-preserving UML model transformations.

Tags

- isUnique: Boolean [1] This tag specifies whether every instance of the concept is unique are or not. If the value is false, then duplicates (i.e. redundant storage of the same object) may exist. The default value is true.

Constraints

- A concept must not have superclasses.
- The upperbound of the multiplicity range of an attribute of a concept must be 1.
- An attribute of a concept must be completely specified by a name and a type.
- The type of an attribute of an concept must be an ADT primitive type or an enumeration type.
- The kind of aggregation that applies to an attribute of a concept is restricted to the value 'composite'.
- The visibility of an attribute of a concept must set to the value 'public'.
- The visibility of an operation of a concept must set to the value 'public'.

Example

Figure 6 shows an ADT-level class diagram of Mesh that contains three concepts: `Mesh`, `Triangle`, and `Vertex`. In this model, every data element (i.e. data record) is stored only once in the data structure of a mesh. Therefore, the tagged value isUnique of all these concepts is set to the boolean value true.

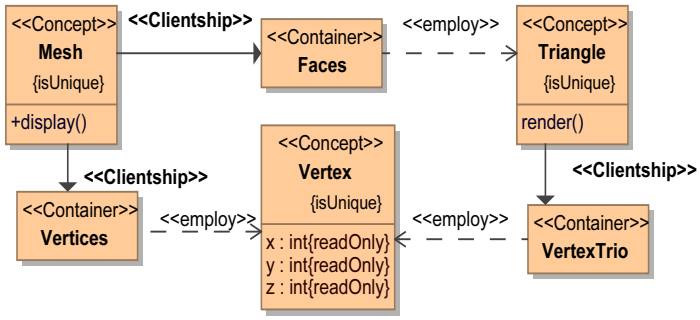

Figure 6: An ADT class diagram of Mesh with the concepts `Mesh`, `Triangle`, and `Vertex`.

2.4 Container

The stereotype Container extends the UML metaclass Class. We introduce this stereotype to show explicitly the collections of data elements at the ADT modeling level. Containers are the classes that represent the entities that collect and store at runtime the multivalued properties (i.e. the attributes and navigable association ends) of the concepts in the conceptual model. A valid ADT class diagram contains no multiplicities with an upper bound larger than one.

We consider only containers that store one type of element (i.e. a Concept). The relationship between a container and its type of elements is indicated by means of the stereotyped dependency <<employ>>.

The tags that are listed below are introduced to indicate the memory-related properties of a container and its elements.

Tags

- capacity: UnlimitedNatural [1] This tag specifies the maximum actual number of elements that are collected in the container. The default value is many ('*').
- isOwningElements: Boolean [1] This tag specifies whether the container owns its elements or whether its elements are owned by another container. The default is true.
- hasUniqueElements: Boolean [1] This tag specifies whether all the elements in the container object are unique or not. The default is true.
- isSharingElementsWith: <<metaclass>> Class [0..*] This tag indicates which types of concepts and containers also have access to the elements of the container. The default is none.
- hasDuplicatesIn: <<metaclass>> Class [0..*] This tag indicates which types of containers store duplicates of the elements of the container. The default is none.
- lifetimeDependency: LifetimeDependencyKind [1] This tag specifies the kind of lifetime dependency that exists between a container and its elements.

For a detailed description of the enumeration type LifetimeDependencyKind we refer to section 2.5. The default is the literal value 'hybrid'.

Constraints

- A container must not have superclasses.
- A container must not be the client of a clientship.
- A container must not employ more than one type of concept.
- The elements of a container must not be owned by more than one container.

Example

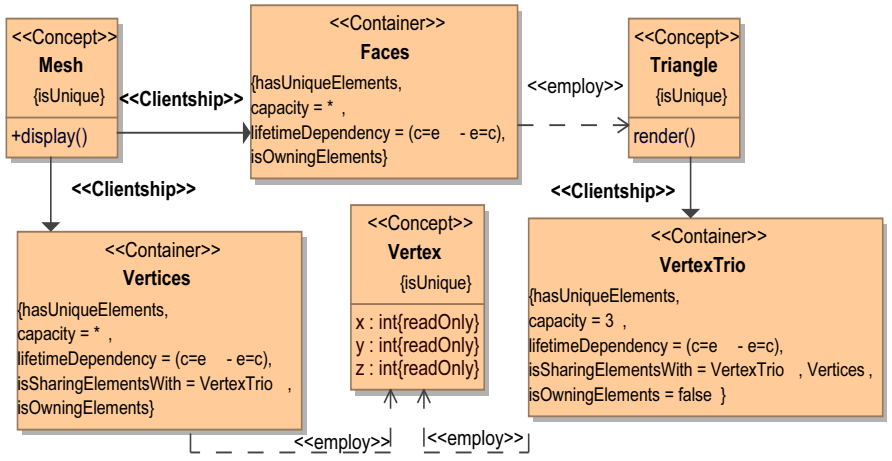

Figure 7: A more detailed ADT class diagram of Mesh.

Figure 7 shows a more detailed version of the ADT model of Mesh introduced in Figure 6. In this ADT model, a `Mesh` concept has access to two containers, namely `Faces` and `Vertices`. `Faces` is a container type that owns its elements (of type `Triangle`). The lifetime of the triangle elements coincides with their container. Every triangle element is unique and is not shared. The `Vertex` elements are gathered in one large owning container of type `Vertices`. Also their lifetime coincides with their container. Every vertex is stored only once. A `Triangle` has access to its three vertices via the container `VertexTrio`, but it does not own them. A vertex has shared access via the containers `Vertices` and `VertexTrio`.

Figure 8 shows an alternative ADT model of Mesh. Here, the `Mesh` concept has access to just one container, namely `Faces`. In this ADT model, a `Vertex` element must not be unique. So, duplicates can be stored as is indicated by the tagged value named *hasDuplicatesIn* in the container `VertexTrio`. However, within a `VertexTrio` the vertices are unique. A `VertexTrio` also owns its three vertices.

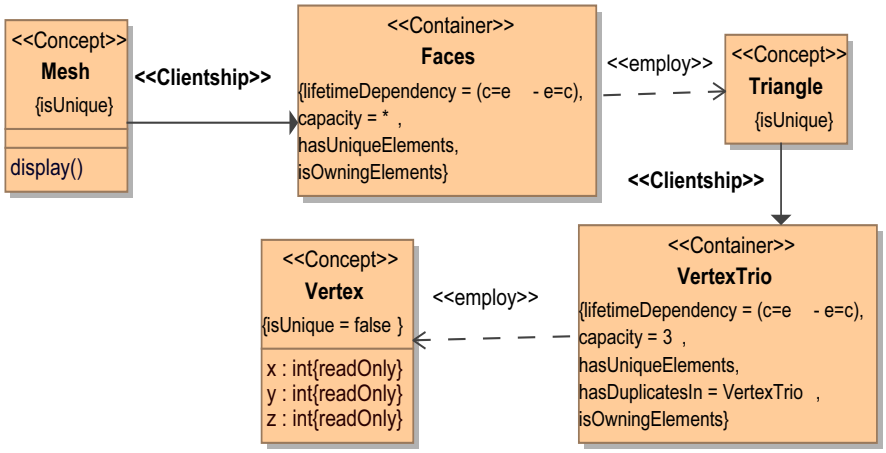

Figure 8: An alternative ADT model of Mesh.

2.5 LifetimeDependencyKind

Lifetime dependency refers to the relation between the lifetimes of a container and its elements. In the UML, the strong form of aggregation (i.e. composition) is defined with coincident lifetime binding between the elements and their container. In our characterization of containership, we consider a weaker form of lifetime binding and allow elements and containers to outlive each other. To characterize the lifetime binding of a container and its elements, we identified two orthogonal properties, namely (i) the temporal relationship (i.e. before, simultaneously, or after) between the creation time of the container and the creation time of the element(s), and (ii) the temporal relationship between the destruction time of the container and the destruction time of the element(s). To limit the number of tag definitions for the stereotype Container, we combined these two lifetime dependency aspects into one single tag named lifetimeDependency. All the possible combinations of these two aspects result in nine different situations of lifetime dependency. We added a tenth possibility to indicate that the elements of a container exhibit different life time bindings (i.e. hybrid).

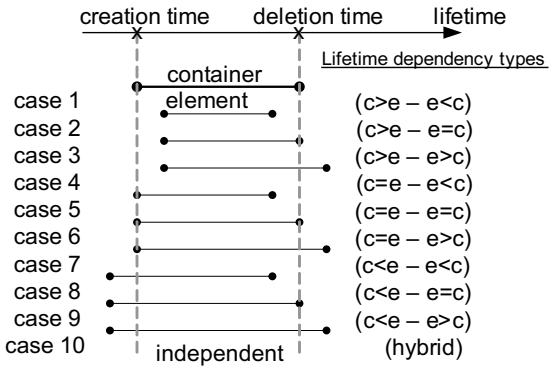

Figure 9: Ten cases of lifetime dependency for the elements and their container.

Thus, we designed LifetimeDependencyKind as an enumeration type that offers ten different values to indicate the kind of lifetime dependency that exists between a container and its elements. Figure 9 shows a sketch in which all the possible configurations associated with the lifetime (i.e. the time from birth to death) of an element are depicted in temporal relation to that of its container. For instance, for *case 1* –denoted as (c>e - e<c)–, the element is created after the container (c>e) and the container is destroyed after the element (e<c).

Lifetime dependency is one of the properties that is ascribed to the Whole-Part relationship (also known as aggregation or composition in UML). Parthood is a relation of significant importance in conceptual modeling. However, there is still much discussion in the conceptual modeling literature on what characterizes this relation (e.g.[8, 9, 10, 11]). In our context, knowing the kind of lifetime dependency is an important fact that can be exploited at ADT-modeling level and also at the concrete data-type level.

2.6 Container Operations

At the ADT modeling level, we want to express the operations of the containers precisely in terms of the *primitive* ADT data-access patterns: i.e., add, remove, select, and traverse. Therefore, we provide the stereotypes add, remove, select, and foreach to indicate that a container operation corresponds to one of these elementary ADT operations. Besides, also the stereotypes isEmpty and COP can be used to model the interface of more *complex* operations of a container. The stereotype Key serves to specify a key that can be used by a <<select>> operation. Finally we present Configuration, which is a stereotype for indicating a specific configuration of the elements inside a container.

2.6.1 add

The stereotype add is a special kind of Operation. It is used to declare that a container operation adds an element to the container. If the adding of the element has to consider a particular configuration of the elements inside the container , then a <<Configuration>> (see Section 2.6.8) can be related to this <<add>>-stereotyped operation by means of a <<configured>> dependency.

Tags

- mode: String [0..1] With this optional tag a modeler can specify a particular mode when adding an element to the container, e.g. ”front” or ”rear”. This information can then be exploited at the concrete data-type level.

Constraints

- The stereotype add must be applied on a container operation.

Example This example is taken from the computer game Snake. Since this game is well-known and rather simple, we assume that this example can be understood easily from the ADT diagram that we present below.

The moving snake object in this game is composed of snake parts. We denote such a part in this ADT model as a snake cell. Hence, the concept `Snake` in the ADT class diagram of model Snake has access to the container `Cells` that owns its snake parts,

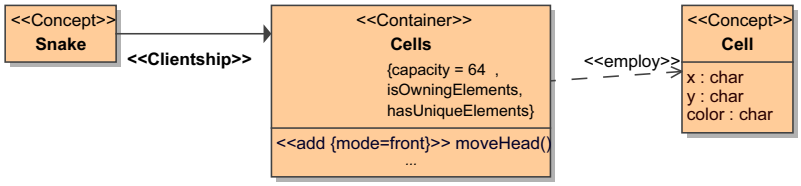

Figure 10: Applying the <<add>> stereotype to the `moveHead` operation in the Snake game.

which are instances of the concept `Cell`. The operation `moveHead` is stereotyped as an <<add>> operation in figure 10. In this way, we indicate that a new snake element must be added in front of the container `Cells` when the snake moves to its next position on the board.

2.6.2 remove

The stereotype remove is a special kind of Operation. It is used to declare that a container operation removes an element from the container. If the removal of the element has to consider a particular configuration of the elements in the container, then a <<Configuration>> (see Section 2.6.8) is related to this <<remove>>-stereotyped operation by means of a <<configured>> dependency.

Tags

- mode: String [0..1] With this optional tag a modeler can specify a particular mode when removing an element from the container, e.g. ”front” or ”largest”. This information can then be exploited at the concrete data-type level.

Constraints

- The stereotype remove must be applied on a container operation.

Example This example also concerns the Snake game. Figure 11 shows that the

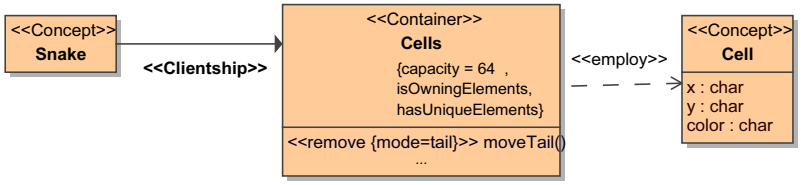

Figure 11: Applying the <<remove>> stereotype to the `moveTail` operation in the Snake game.

operation named `moveTail` is stereotyped as a <<remove>> operation. The snake element at the rear end of the container `Cells` must be removed from the container `Cells` when the snake moves to its next position on the board.

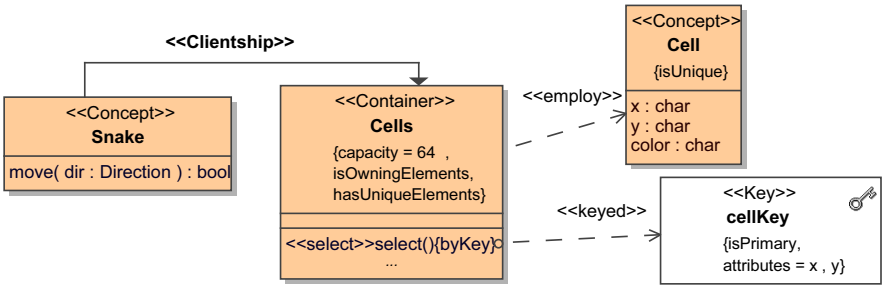

Figure 12: Example of a < <select> > stereotyped operation .

2.6.3 select

The stereotype select is a special kind of Operation. It is used to declare that a container operation selects an element (or elements) from the container. This selection can happen by key. In that case, the < <select> >-stereotyped operation must be related by a < <keyed> > dependency to a < <Key> > (see Section 2.6.7). If no key or mode is specified, then all the data of the element (which should be passed as a parameter of the operation) must be taken into consideration for the identification of the selected element(s).

Tags

- mode: String [0..1] With this optional tag a modeler can specify a particular mode to select an element(s) from the container, e.g. ”second” or ”oldest”. This information can then be exploited at the concrete data-type level.
- /byKey: Boolean [1] This tag specifies whether the select operation is steered by a key (cf. Section 2.6.7). The default is true.

Constraints

- The stereotype select must be applied on a container operation.

Example Figure 12 shows an example of the use of the < <select> > stereotype. In the Snake game, the `select` operation of the container `Cells` is stereotyped as < <select> > to declare that this operation selects an elements from the container. This selection is steered by a key, which is indicated by the tagged value named byKey. The key is modeled by means of a Key stereotype named `cellKey`.

2.6.4 foreach

The stereotype foreach is a special kind of Operation. It is used to declare that a container operation traverses the elements of the container to perform some action on these elements.

Tags

- operation: < <metaclass> > Operation [1] This tag specifies which operation has to be applied on the elements of the container during the traversal.

Constraints

- The stereotype foreach must be applied on a container operation.
- The value of the tag named operation must be an operation that is owned by the concept, which is employed by the container.

Example

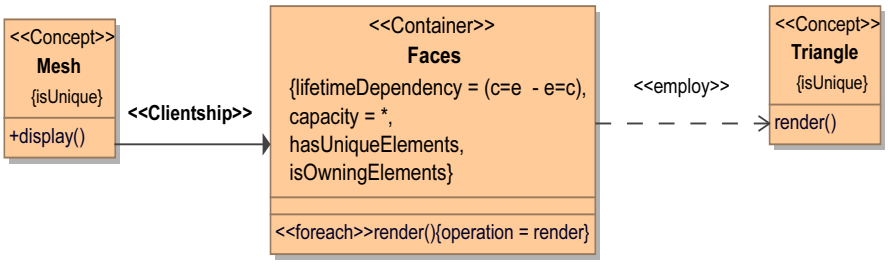

Figure 13: Example of specifying a container operation as < <foreach> >, i.e. the operation `render`() of container `Faces` requires the traversal of the collection of triangles to display a mesh.

To display a `Mesh` object on the screen, every `Triangle` element in the container `Faces` has to be rendered. Figure 13 shows how the `render`() operation of the container `Faces` is stereotyped as a < <foreach> > operation. The tagged value named operation has the value `render`(), which refers to the `render`() operation of the concept `Triangle`.

2.6.5 isEmpty

The stereotype isEmpty is a special kind of Operation. It is used to declare that a container operation checks for the presence of elements in a container. The return type is Boolean.

Tags no tags

Constraints

- The stereotype isEmpty must be applied on a container operation.

Example We refer to the example of the COP operation in the following section.

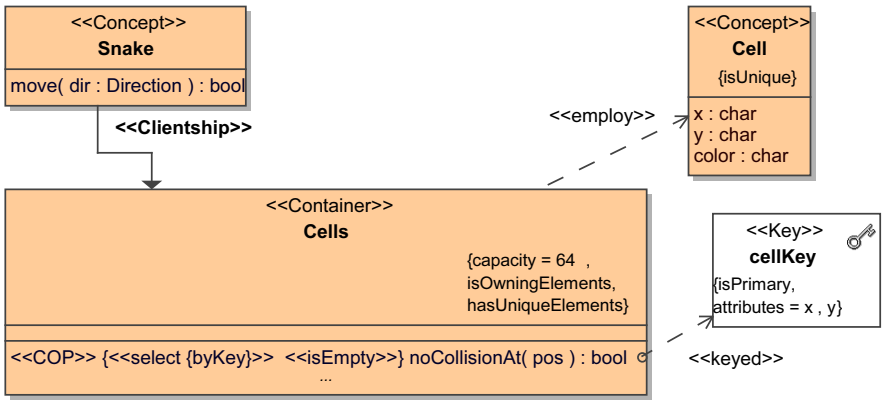

Figure 14: Example of using the COP operation stereotype.

2.6.6 COP

The stereotype COP is a special kind of Operation. COP is an acronym for Composite OPeration. For this stereotype, we are inspired by the programming language literature, more specifically "Higher Order Messaging" [12] and the "Higher Order Functions" from the functional languages and some object-oriented programming languages like Smalltalk. A COP operation specification allows a compact notation for a complex container operation that is composed as a sequence of container operations.

Tags

- expr: < <metaclass> > Operation [1..*] {ordered, nonunique} This tag specifies the sequence of container operations that form the composite container operation. The evaluation[2] of this expression proceeds from left to right.

Constraints

- The stereotype COP can only be applied on a container operation.
- The values of the tag named expr must be operations of the container class that owns this COP operation.

Example

Figure 14 shows an example that also comes from the Snake application.
The container operation `noCollisionAt(pos)` is a composite operation, which is indicated by means of the stereotype COP. The expression for this operation is formulated via the tagged value named expr as follows: < <select {byKey}> > < <isEmpty> >. This means that the collision detection of the moving snake with its own body consists in checking whether the snake cell at position `pos` on the snake board belongs to the container `Cells` or not. If the set of selected snake cells is empty, then no collision occurs.

[2]The evaluation happens when this operation is called by sending a message to the container.

2.6.7 Key

To select an element (or a set of elements) from a container, some applications need a key to specify which element(s) should be selected. For that purpose we provide the stereotype Key. A key is specified as an ordered list of these attributes of the concept that compose the key (see tagged value named attributes). In an ADT class diagram, the <<select>> operation that uses a key is related to this key by means of a dependency that is stereotyped as <<keyed>>.

Specifying a key has to be considered as providing (exploitable) semantic application-specific information. As such, it does not imply any restriction on the implementation at the concrete data-type level.

Tags

- attributes: <<metaclass>>Property [1..*] {ordered} This tag specifies the sequence of attributes that form the key that is used to select an element from a container.
- isPrimary: Boolean [1] This tag specifies whether the key is a primary or not. The default value is true.

Constraints

- The values of the tag named attributes must be properties owned by the concept that is employed by the container that owns the <<select>> operation.

Notation

In an ADT class diagram, a key is shown using the classifier symbol. In the upper right corner of the rectangle, a small picture of a key is placed. Additionally, the name of the key can be preceded by the stereotype indication <<Key>>. Figure 15 shows an example of a key named `cellKey`.

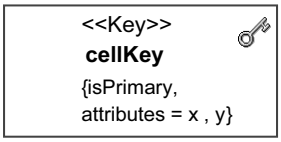

Figure 15: The notation for a <<Key>>.

Example Figure 12 shows an instance of the Key stereotype in the ADT model of the Snake game. The <<Key>> named `cellKey` is composed of the attributes `x` and `y` (i.e. the coordinates or position of a cell on the board) of the concept `Cell`. This primary key allows to uniquely identify a cell in the snake. The `select` operation of container `Cells` uses this key, which is indicated by the <<keyed>> dependency.

2.6.8 Configuration

The stereotype Configuration allows a modeler to indicate that the elements in a container exhibit a specific configuration. This means that there exists a structural or functional relationship between the elements inside the container. Operations that change or rely on the configuration of a container (e.g., <<add>>, <<remove>>, or <<foreach>> operations) can be related to a <<Configuration>> by means of a <<configured>> dependency.
Configuration is considered in the conceptual modeling literature as one of the secondary properties of the Whole-Part relationship (e.g. [10]).

Tags

- kind: ConfigurationKind [1] This tag is used to specify the kind of configuration, i.e. *functional* or *structural*. The enumeration type ConfigurationKind provides these literal values.
- mode: String [1] With this tag a modeler specifies the semantics of the configuration relationship in words.

Constraints no constraints

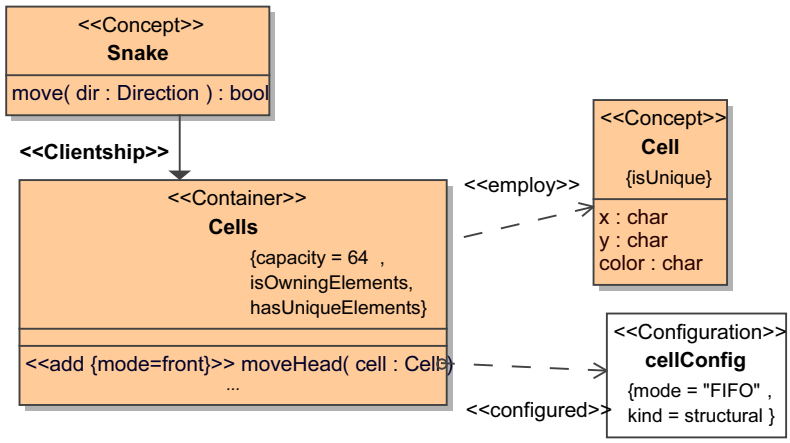

Figure 16: Example showing the container of snake cells, which is modeled as a FIFO configuration.

Example Figure 16 shows an example of a Configuration instantiation. In the ADT class diagram of the Snake structure, the <<Configuration>> named `cellConfig` indicates that the elements of the container `Cells` are structurally related in a FIFO configuration. This information is used by the `moveHead` operation, which adds a new cell in front of the collection of snake cells.

3 ADT Primitive Types Model Library

The content of the ADT Primitive Types model library is shown in Figure 17. A number of primitive data types have been defined for use in the ADT models. These include the fixed-point data types of the source-code level (i.e. C/C++), such as char or int.

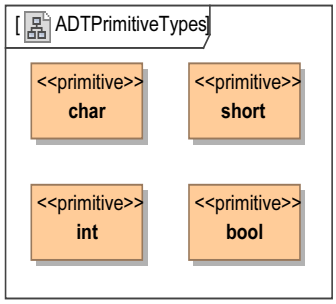

Figure 17: Contents of the ADT Primitive Types model library.

The reference platform

The ADT Primitive Types library provides us with the basic data types that represent the built-in atomic data types of a reference platform. We introduce this reference platform to abstract away from concrete platform-related characteristics and to enable the estimation of the cost metrics at the modeling level in a platform-independent way.

3.1 char

A char type represents the built-in type char from the source-code level. On the reference platform, a char consumes one byte of memory space. It is used for attributes and expressions in the ADT model whose value can be encoded with maximally eight bits.

Attributes No additional attributes

Constraints No additional constraints

3.2 short

A short type is used to represent the source-code level primitive type short. On the reference platform, a short consumes two bytes of memory space. It is used for attributes and expressions in the ADT model whose value can be encoded with maximally 16 bits.

Attributes No additional attributes

Constraints No additional constraints

3.3 int

An int type represents the source-code level primitive type int. On the reference platform, an int consumes four bytes of memory space. It is used for attributes and expressions in the ADT model whose value can be encoded with maximally 32 bits.

Attributes No additional attributes

Constraints No additional constraints

3.4 bool

A bool type is used to represent the source-code level primitive type bool. On the reference platform, a bool consumes one byte of memory space.

Attributes No additional attributes

Constraints No additional constraints

Enumeration Types

The ADT profile allows user-defined data types as instances of the metaclass Enumeration. On the reference platform, the memory footprint of an enumeration amounts to the size of the smallest primitive type that can hold the range of the set of enumeration literals. For instance, an instance of an enumeration type with maximally 256 literals consumes just one byte in memory.

Pointer Types

The ADT profile provides no reference or pointer types. A Pointer, as a modeling concept, does not belong to the ADT abstraction level. In an ADT model, pointers are abstracted to clientship relations or are encapsulated inside the containers by means of Configurations. The refinement of the clientships and containers is the responsibility of the ADT- implementation step. Here, in the final source-code generation step, will be decided where pointers are introduced.

Example

In Figure 18, the concept `Vertex` is modeled with three integer attributes, namely the coordinates `x`, `y` and `z`. Hence, the memory footprint of a vertex amounts to 3 x 4 bytes = 12 bytes on the reference platform.

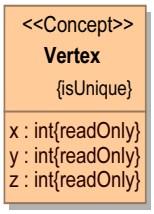

Figure 18: The concept `Vertex` modeled with three integer attributes

4 ADT Behavior Profile

The goal of the ADT Behavior profile is to provide the designer with modeling elements that can be used to express exploitable application-specific knowledge of the data of the application at the ADT modeling level. For now, this profile contains one single stereotype named Clustered and one enumeration type Clusterkind. In these modeling elements we gathered the exploitable data properties experienced in our case studies. Consequently, this profile covers just the part of the set of characteristics identified in the investigated application domain. We are aware that this profile is not complete. Additional experiments in other application domains, which is one of the issues of future research, is needed for the completion of this profile. Figure 19 shows the contents of this profile.

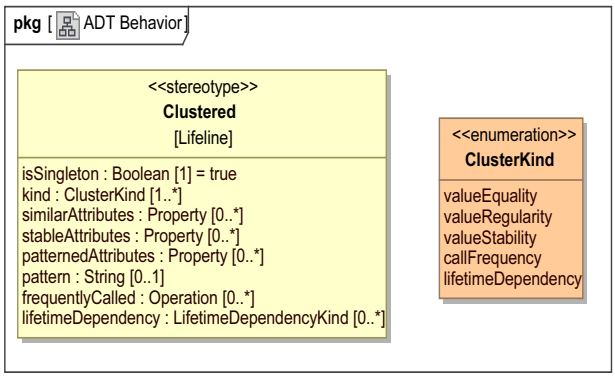

Figure 19: Contents of the ADT Behavior profile.

4.1 Clustered

The stereotype Clustered is a special kind of Lifeline (cf. [1]). This stereotype serves to annotate the lifeline of a container in a sequence diagram with extra information (i.e. meta-data) on the run-time characteristics of the elements in the container and this for the scenario that is subject of the optimization. This application-specific information can then be exploited during the optimization of the ADT model.

In this profile, a container can only contain elements of the same type (i.e. < <Concept> >). However, at run time, these elements often behave in different ways. Moreover, we can

observe groups of elements with similar behavioral patterns. We call such a group a *cluster*. We refer to the research work of Mitchell and Power [13] that reports on measuring clustering in Java programs.

Because we focused on dynamic behavior of the collections of data objects, we identified in our case studies several types of clustering:

- clustering based on *value stability* of the attributes of the elements: i.e., the value of the attributes does not change, and can thus be considered as an invariant [14] for that cluster.
- clustering based on *value similarity* of the attributes of the elements: i.e. the value of the attributes changes simultaneously, and thus be considered as a common variable for the cluster.
- clustering based on a *regular pattern* in the values of the attributes. If this regularity is stable, then this can be exploited (e.g. by compression).
- clustering based on the *similarity in the kind of operations* that are called on the elements. Also this behavior can be exploitable.
- clustering based on the *similarity in lifetime dependency* of the elements. E.g., by isolating a cluster of short-living elements in a separate container, we can reduce the memory footprint of the original data structure.

If a container object exhibits clustering, then the designer can annotate the lifeline of the container in the sequence diagram of the scenario by means of the < <Clustered> > stereotype.

Tags

- isSingleton: Boolean [1] This tag is used to specify whether there exists only one cluster in the container or more. The default value is true.
- kind: ClusterKind [1..*] This tag is used to specify the kind of cluster(s) that exists in the container object. The enumeration type ClusterKind provides for that purpose the following literal values: i.e., *valueStability*, *valueEquality*, *valueRegularity*, *callFrequency*, and *lifetimeDependency*.
- stableAttributes: < <metaclass> > Property [0..*] With this tag a modeler can specify which attributes of the concept exhibit the *valueStability* clustering type.
- similarAttributes: < <metaclass> > Property [0..*] With this tag a modeler can specify which attributes of the concept exhibit the *valueRegularity* clustering type.
- patternedAttributes: < <metaclass> > Property [0..*] With this tag a modeler can specify which attributes of the concept exhibit the *valueEquality* clustering type.
  - pattern: String [0..1] The value of this tag gives a description of the pattern that is exhibited by the regularity in the values of the attribute(s) in the cluster. E.g., the values of the attributes could form an interval ranging from a certain start value to a certain end value.

- frequentlyCalled: < <metaclass> > Operation [0..*] This tag is used to list the names of the operations for the *callFrequency* clustering type.
- lifetimeDependency: lifetimeDependencyKind [0..*] This tag is used to list the kind(s) of lifetime dependency for the cluster(s) that exhibit *lifetimeDependency* clustering.

Constraints no constraints

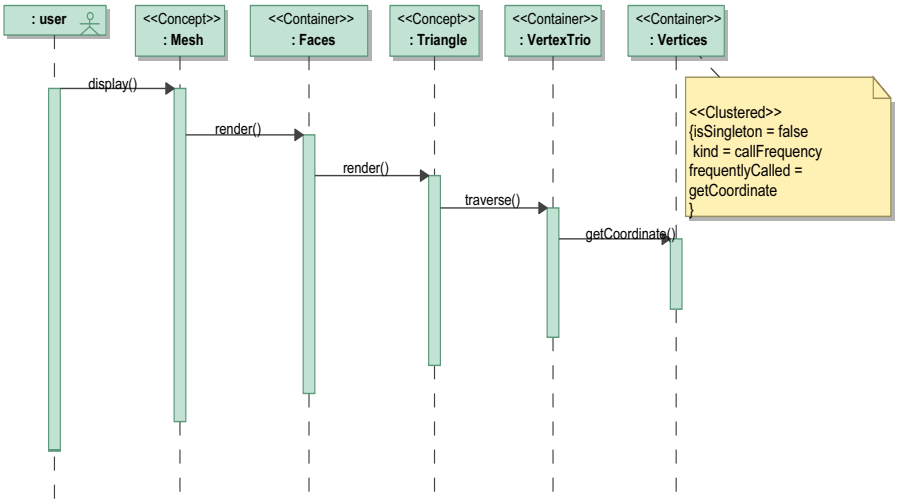

Figure 20: Annotation of the `display` scenario for the Mesh rendering. The lifeline of container `Vertices` is annotated as < <Clustered> > because the three vertices of a triangle are always accessed in group when getting the values of their coordinates.

Example Figure 20 shows the sequence diagram of the `display` scenario of the Mesh application. This sequence diagram conforms to the data model that is depicted in Figure 7 in which all the vertices are stored in one large container of type `Vertices`. In the sequence diagram, the lifeline of the `Vertices` container object is annotated as < <Clustered> > because the three vertices of a triangle object are always accessed together to get the values of their coordinates during the rendering process. We exploited this application-specific knowledge to obtain the data model that is showed in Figure 8.

References

[1] OMG, “Unified modeling language: Superstructure specification, v2.1.1,” http://www.omg.org/docs/formal/07-02-03.pdf, February 2007.

[2] C. Bunse, “Pattern-based refinement and translation of object-oriented models to code,” Ph.D. dissertation, Univ. of Kaiserslautern, Kaiserslautern, Germany, September 2000.

[3] C. Bunse and C. Atkinson, "The Normal Object Form: Bridging the gap from models to code." in *UML '99: Proceedings of the Second International Conference on The Unified Modeling Language*, 1999, pp. 691–705.

[4] A. V. Aho, J. E. Hopcroft, and J. Ullman, *Data Structures and Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[5] E. G. Daylight, D. Atienza, A. Vandecappelle, F. Catthoor, and J. M. Mendías, "Memory-access-aware data structure transformations for embedded software with dynamic data accesses," *IEEE Trans. VLSI Syst.*, vol. 12, no. 3, pp. 269–280, 2004.

[6] M. R. Blaha and W. J. Premerlani, "A catalog of object model transformations." in *WCRE*, 1996, pp. 87–96.

[7] K. Lano and J. Bicarregui, "Semantics and transformations for UML models," in *UML '98: Selected papers from the First International Workshop on The Unified Modeling Language UML'98*, 1999, pp. 107–119.

[8] J. Odell, "Six different kinds of composition," *Journal of Object-Oriented Programming*, vol. 5, no. 8, pp. 10–15, January 1994.

[9] M. Saksena, R. France, and M. M. Larrondo-Petrie, "A characterization of aggregation," *International Journal of Computer Systems Science and Engineering*, vol. 14, no. 6, pp. 363–371, 1999.

[10] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel, "Formalization of the whole-part relationship in the unified modeling language," *IEEE Trans. Softw. Eng.*, vol. 29, no. 5, pp. 459–470, 2003.

[11] G. Guizzardi, "Ontological foundations for structural conceptual models," Ph.D. dissertation, University of Twente, Enschede, The Netherlands, October 2005.

[12] M. Weiher and S. Ducasse, "Higher order messaging," in *DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium*, 2005, pp. 23–34.

[13] A. Mitchell and J. F. Power, "Using object-level run-time metrics to study coupling between objects," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 1456–1462.

[14] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 259–269.

[15] MagicDraw, http://www.magicdraw.com.

[16] LORE-Website, http://www.lore.ua.ac.be/Research/Artefacts/.