An MDA approach towards integrating formal and informal modeling languages

Soon-Kyeong Kim, Damian Burger, David Carrington

Institutions: University of Queensland

Published on: 18 Jul 2005 - Formal Methods

Topics: Modeling language, Formal language, Formal methods, Model transformation and Unified Modeling Language

An MDA Approach Towards Integrating Formal and Informal Modeling Languages

Soon-Kyeong Kim, Damian Burger, and David Carrington

School of Information Technology and Electrical Engineering,
The University of Queensland, St. Lucia, 4072, Australia
{soon, damian, davec}@itee.uq.edu.au

Abstract. The Model Driven Architecture (MDA) involves automated transformations between software models defined in different languages at different abstraction levels. This paper takes an MDA approach to integrate a formal modeling language (Object-Z) with an informal modeling language (UML) via model transformation. This paper shows how formal and informal modeling languages can be cooperatively used in the MDA framework and how the transformations between models in these languages can be achieved using an MDA development environment. The MDA model transformation techniques allow us to have a reusable transformation between formal and informal modeling languages. The integrated approach provides an effective V&V technique for the MDA.

1 Introduction

Integration between formal and informal or semi-formal visual modeling (or specification) languages is a well-known topic in the literature [8, 11, 12, 14]. There are many advantages to be gained from integrating formal techniques with informal or semi-formal approaches in the field of software development. Integration can make formal methods easier to apply and informal methods more precise, aiming towards "the best of both worlds". Despite the potential for taking benefits from both types of techniques, the integrated approach is seldom used in practice. Several drawbacks we have identified are: transformations between formal and informal models are often not explicitly defined [1, 8, 13, 14, 15, 23], which makes it difficult to know on what semantic basis the transformation has taken place, whether semantics of models are preserved during the transformation and whether the transformation is complete and consistent. Also a lack of tool support for the actual transformation is a drawback in this area. In order to contribute to this area, this paper presents an MDA approach towards the integration of a formal modeling language Object-Z [4] with the Unified Modeling Language (UML) [19], a semi-formal visual modeling language.

The Model Driven Architecture (MDA) [18] is a new software development framework that aims to separate business logic from underlying platform technology. It involves automated transformations between software models defined in different languages. In MDA, a Platform Independent Model (PIM) of a system is specified and a Platform Specific Model (PSM) is derived from the PIM using transformations. MDA model transformation can be applied to the integrated approach. In the MDA, models are integrated by their common basis in the Meta Object Facility (MOF) [16], which is the meta-language standard for UML and the other OMG modelling languages. That is, each modeling language is defined in terms of a metamodel using the MOF. Given the metamodels of different modeling languages, a set of transformation rules is defined explicitly using a transformation language, which is also a MOF model. Actual transformations are then achieved automatically using a transformation tool that understands the transformation language. In this paper, we use this reusable MDA transformation framework for modeling language integration with Object-Z and UML. For this, we first define Object-Z in terms of a metamodel based on the MOF. Given the metamodels of UML and Object-Z, we then define transformation rules using a transformation language[1]. The metamodel-based MDA transformation framework allows us to define transformations precisely and explicitly in terms of transformation rules, which is critical for rigorous model evolution from informal to formal and vice-versa. It also allows us to have a reusable transformation that can be applied to any models in the two languages. Actual transformations are achieved using tools supporting MDA.

Additionally our integrated approach can deliver benefits to MDA. To get the full potential of the MDA, the MDA transformation infrastructure (currently being standardized [18]) should include the ability to use modelling notations that are the most appropriate to capture different aspects of a system, and should have a capability of transforming between models in these different notations. Also there must exist efficient ways to check the models for properties such as consistency and correctness. Currently UML is proposed as the central modelling language by OMG in the MDA. However, using only UML has limitations to provide these capabilities required for the MDA. Our integrated approach with formal and informal modeling techniques can contribute to this area. For example, it provides the convenience to choose appropriate modeling techniques to capture different aspects. Formal techniques provide effective means to check models providing increased quality for both specification and implementation. In this integrated MDA modeling framework, models are corrected and evolved via model transformation from informal to formal and vice-versa. In fact, the integrated approach can be a V&V technique for the MDA. For example, an Object-Z model derived from a UML model is a V&V model of the UML model. Any formal reasoning techniques available for Object-Z can be used to validate the UML model.

It should be noted that in this paper it is not our intention to present a complete definition of Object-Z or UML, or a complete set of transformation rules between the two languages. Rather we focus on explaining how the MDA model transformation framework can be applied to the integration of the two languages. The transformation presented in this paper should pave the way for others to follow the same transformation approach towards integrating different formal and informal modeling languages.

The structure of the rest of this paper is as follows. Section 2 discusses relevant background information. Section 3 presents the model transformation environment

[1] In this paper, we use the Distributed Systems Technology Centre (DSTC)'s transformation language [3] that has been submitted to respond to the OMG's MOF 2.0 Query/Views/Transformations (QVT) Request for proposals [17], and its transformation engine Tefkat [2]. Once the OMG finalizes a standard transformation language, the transformation rules can be converted into the standard language.

used in this work and its rationale. Section 4 discusses the transformation itself in detail with an example. Finally, Section 5 concludes and discusses further work.

2 Background Information

In this section, we present a metamodel of Object-Z and a metamodel of UML. The Object-Z metamodel presented in this section is an enhanced version of the one presented in [9, 10]. The UML metamodel presented in this section is a simplified version of UML 2.0 [19].

2.1 Object-Z Metamodel

Figure 1 is a UML class diagram showing core model elements in Object-Z and their structure (we add OZ to the names of the model constructs to distinguish them from the UML modeling constructs). Figure 2 shows types in Object-Z (see [10]).

OZElement is a top-level metaclass from which all possible model elements in Object-Z can be drawn. OZNamedElement represents all model elements with names (e.g. attributes, classes, operations, and parameters). OZNamespace is an element that can own other named elements (e.g. classes or operations).

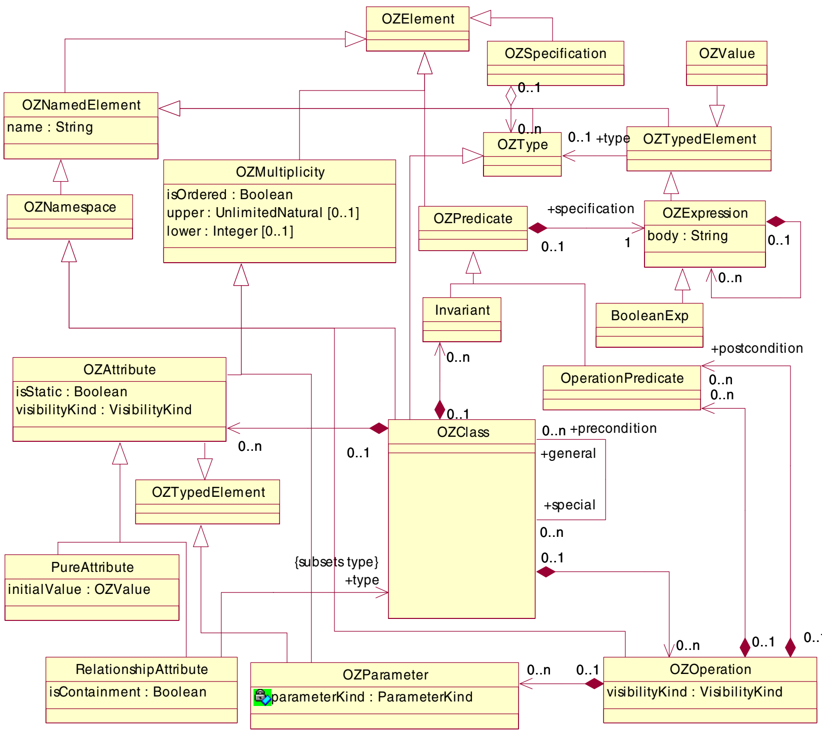


Fig. 1. Object-Z model elements

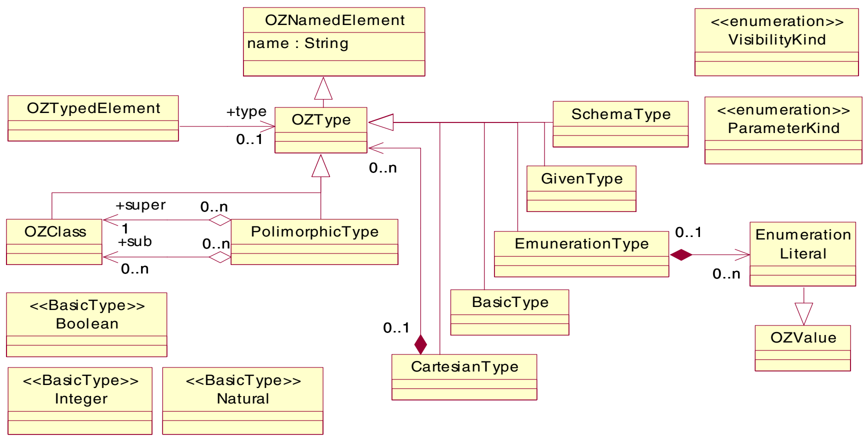


Fig. 2. Object-Z types

OZTypedElement presents elements with types (e.g. attributes and parameters) and the type of a typed element constrains the set of values that the typed element may refer to. OZMultiplicity is an abstract metaclass of elements with multiplicity information that specifies the allowable cardinalities for an instantiation of the elements (e.g. attribute values and parameter values). It also has an attribute (isOrdered) to define whether the values in an instantiation of this element must be ordered.

OZPredicate is a metaclass to define a condition in classes or operations. Conditions defined in a class are invariants and conditions defined in an operation are either a precondition or a postcondition of the operation. Predicates contain expressions that will have a set (possibly empty) of values when evaluated in a context. Boolean expressions are one type of expression in Object-Z.

UML is a visual modelling language and does not provide a language for specifying expressions al-though OCL [19] is recommended as a constraint language for UML by OMG. Consequently UML treats expressions as an uninterpreted textual statement (see the meta-class OpaqueExpression in the UML metamodel) and the semantics of expressions depends on the language. For this reason, we do not further clarify expressions in Object-Z in this paper focusing on transforming the structural constructs of the two languages and leave this issue as further work to map Object-Z expressions to a specification language such as OCL.

Classes: In Object-Z, classes are the major modeling construct for specifying a system. A Class is a template for objects that have common behaviors. A Class has a set of attributes (PureAttibute) and a set of operations. Each attribute has a name, a type, a visibility and an attribute (isStatic) specifying whether the attribute is static. By specializing multiplicity element, an attribute supports a multiplicity that specifies valid cardinalities for the set of values associated with the attribute. An operation has a name, a visibility and a set of parameters, each of which also has a name, a type and the multiplicity information for the set of values associated with the parameter.

Relationships and instantiation: Classes can be instantiated in other classes as attributes. In Object-Z, instantiation is used as a mechanism for modeling relationships between objects, which in UML is modeled using a separate modeling construct, Association. Objects that instantiate other classes as their attributes (RelationshipAttribute) can refer to the objects of the instantiated classes. The values of these attributes are object-identities of the referenced objects. Each relationship attribute has an attribute (isContainment) specifying whether the referenced objects are owned by their referencing object (a containment relationship).

Inheritance: Classes in Object-Z can be used in defining other classes by inheritance. A class can inherit from several classes (multiple inheritance). In the Object-Z metamodel, inheritance is defined with an association between classes.

2.2 A Simplified UML Metamodel

Figure 3 presents class modeling constructs in UML. In this paper, we are concerned with only a subset of the UML modeling constructs that are relevant to the discussion of transformation with Object-Z. For a full description of the UML 2.0 metamodel refer to [19].

Classes: A class in UML is a descriptor of a set of objects with common properties in terms of structure, behavior, and relationship. Class is a kind of classifier whose features are attributes and operations. Attributes of a class are represented by instances of Property that are owned by the class. Some of these attributes may represent the navigable ends of binary associations. An attribute has a name, a visibility, a type, and amultiplicity. An operation also has a name, a visibility and parameters. Each parame-

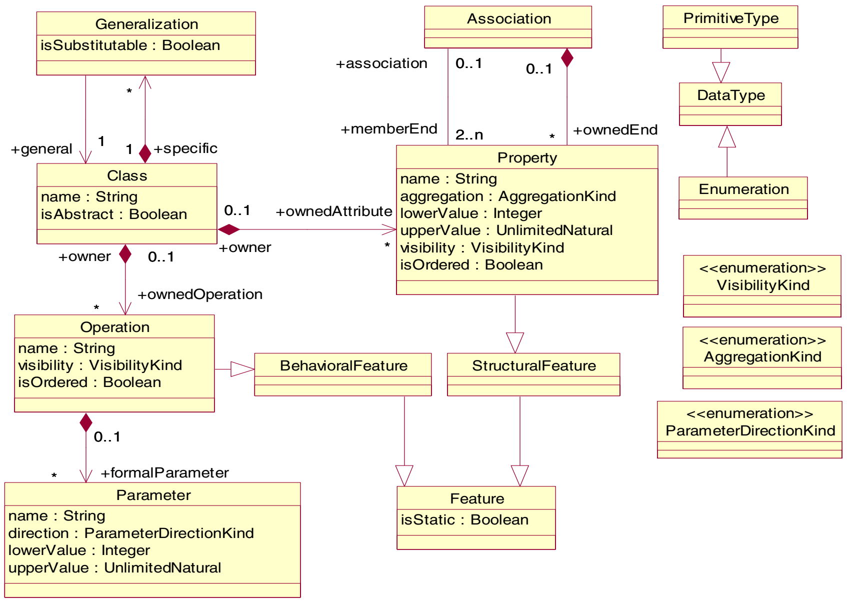


Fig. 3. A simplified UML metamodel

ter of an operation has a name and a given type. Attributes and operations have a visibility. Visibility in UML can be private, public, or protected.

Associations: In UML, relationships between classes are represented as associations. An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. When a property is owned by an association, it represents a non-navigable end of the association. In this case the property does not appear in the namespace of any of the associated classifiers. When a property at an end of an association is owned by one of the associated classifiers, it represents a navigable end of the association. In this case the property is also an attribute of the associated classifier. Only binary associations may have navigable ends. A property of an association has attributes indicating whether the property has an aggregation (aggregation) and if it is compositionally aggregated (isComposite).

Generalizations: In UML, a generalization is a taxonomic relationship between a more general class and a more specific class. Each instance of the specific class is also an indirect instance of the general class. Thus, the specific classifier inherits the features of the more general class. An attribute, isSubstitutable, indicates whether the specific class can be used wherever the general class can be used.

3 Transformation Environment

Sendal and Kozaczynski [20] identify a number of challenges in model transformation. Most importantly, defining a model transformation requires a clear understanding of the abstract syntax and the semantics of both the source and target models. In the metamodel-based approach, each modelling notation is precisely defined in terms of its metamodel (using the OMG's MOF). Model transformations are then defined in terms of the relationship between a source MOF metamodel and a target MOF metamodel. Previously the authors defined a set of formal mapping functions between Object-Z and UML 1.4 based on their metamodels [9]. We implement these formal mapping functions using a transformation language in a MDA development environment. This section covers background information of the implementation.

3.1 DSTC Transformation Language Overview

In 2002, OMG issued a Queries, Views and Transformations (QVT) Request For Proposals (RFP) [17] and is currently in the process of selecting a standard MDA model transformation language. Several proposals have been submitted to the request. The Distributed Systems Technology Centre (DSTC)'s transformation language [3] is one of them and is used in this paper to define mappings between UML and Object-Z.

Figure 4 illustrates how an Object-Z model is transformed into a UML model using the DSTC's transformation language. At the meta-level, we have four metamodels: the UML metamodel, an Object-Z metamodel, a Transformation metamodel defining the concepts in the DSTC's trans-formation language and a Tracking metamodel defining the mapping relationships between model elements in Object-Z and UML. The diagram in Figure 5 is a Tracking metamodel used in our work.

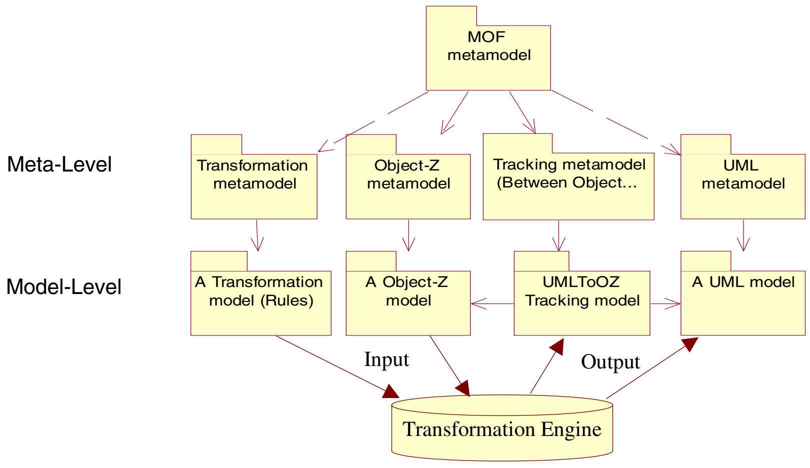


Fig. 4. Model Transformation with the DSTC's Transformation Language

For example, an Object-Z class maps to a UML class, an Object-Z attribute maps to a UML attribute, an Object-Z operation maps a UML operation and an Object-Z attribute modelling a relation maps to an association in UML. Since both the languages share common concepts in object technology, mappings between these two languages are mostly straightforward.

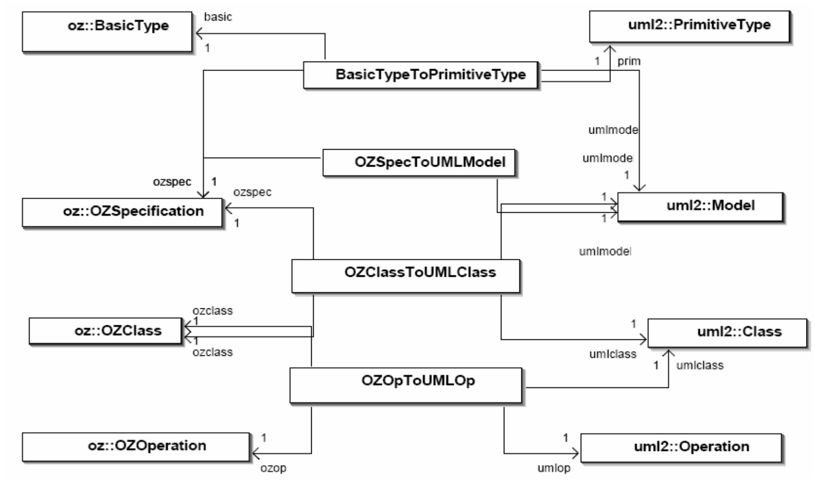


Fig. 5. Tracking model for UML and Object-Z

At the model-level, an Object-Z model (an instance of the Object-Z metamodel) and a transformation model are input to a transformation engine. The transformation model includes a set of transformation rules specifying how to convert an Object-Z element into a UML model element. These rules are based on the mapping relationships defined in the Tracking metamodel. Then the transformation engine populates a target UML model based on the source model according to the transformation rules. During the transformation, a tracking model is created and used to store correspondences between source elements and target elements. These correspondences can then be used to link transformation rules together. For example, the OZClassToUMLClass tracking class records the corresponding UML class for each Object-Z class. This is stored so that the UML class generated from a particular Object-Z class can be looked up from other transformation rules. For example, a UML operation generated for an Object-Z operation can be inserted into the right UML class by querying the tracking model.

The DSTC's transformation language consists of three major concepts: pattern definitions, transformation rules, and tracking relationships [3]:

- Pattern definitions can be used to define common structures to be reused throughout a transformation.
- Transformation rules are used to describe the elements to be created in a target model based on the elements in a source model. Transformation rules can be extended or superseded allowing for modularity and reusability.
- Tracking relationships are used to associate target elements with the source elements that led to their creation, important for rule reuse.

Currently the DSTC's Transformation language uses a concrete syntax in the style of SQL [3]. An example transformation rule (OZSpec2UMLModel) in the DSTC transformation language is given below. It simply maps each Object-Z specification to a UML Model. Line 1 declares the rule name and variables to be used in the rule. Lines 2 and 3 then express that for each Object-Z specification found in the source model (FORALL statement), a UML Model should be created in the target model (MAKE statement). Line 4 preserves the tracking relationship between the UML Model that was created and the Object-Z specification it was created from. This is done by using a LINKING...WITH statement and setting the ozspec and umlmodel references of the tracking model class OZSpecToUMLModel (see Figure 5). This tracking will allow other rules to find the corresponding UML Model for an OZ specification. We present other rules in detail in Section 4.

```
1    RULE OZSpec2UMLModel(ozs, umlm)
2    FORALL  OZSpecification ozs
3    MAKE    Model umlm
4    LINKING OZSpecToUMLModel WITH ozspec = ozs, umlmodel = umlm;
```

3.2 Model Transformation Tool Environment

Figure 6 shows the overall tool architecture used in our work. The Eclipse Platform [5] is a universal tool platform – an IDE that allows tool developers to add functionality through tool plug-ins. It is used as a tool integration environment for transformation. The plug-ins we use are: EMF [7] and Tefkat [2].

3.2.1 Eclipse Modeling Framework (EMF)

EMF is a Java framework for building applications based on simple class models [7]. It allows developers to turn the models into customizable Java code. EMF plays a very important role in our transformation tool architecture as it is used to construct the metamodels and instances that are used as input to the Tefkat Transformation Engine.

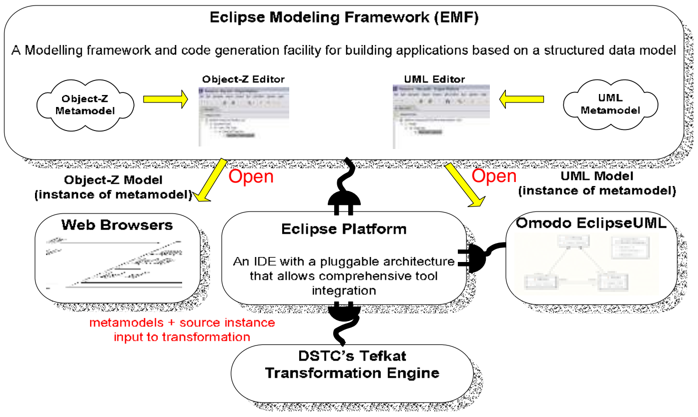

Eclipse Modeling Framework (EMF)
A Modelling framework and code generation facility for building applications based on a structured data model
Object-Z Metamodel
Object-Z Editor
UML Editor
UML Metamodel
Object-Z Model (instance of metamodel)
Open
Open
UML Model (instance of metamodel)
Web Browsers
Eclipse Platform
An IDE with a pluggable architecture that allows comprehensive tool integration
Omodo EclipseUML
metamodels + source instance input to transformation
DSTC's Tefkat Transformation Engine

Fig. 6. Tool Architecture used in this work

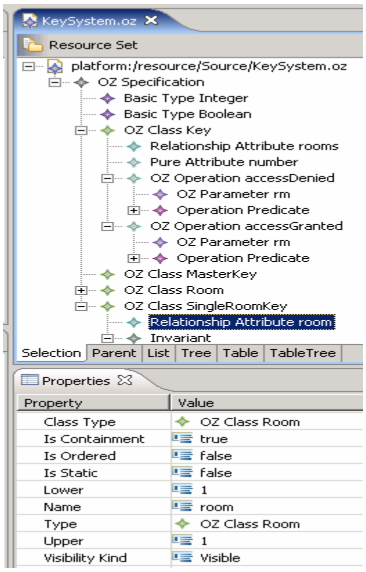
The metamodels for Object-Z and UML defined using EMF are Ecore models (e.g. the Object-Z metamodel Object-Z.ecore, the UML metamodel UML2.ecore[2], and the Tracking metamodel OZToUMLTracking.ecore), which then allows the automatic generation of an editor to create Object-Z (instance) models, stored in XMI format, to be transformed into UML. The Ecore language used to create models in EMF is a core subset of the OMG's MOF [16] that provides a common basis of models in the MDA. However, to avoid any confusion, the MOF-like core meta model in EMF is called Ecore. In fact, EMF can generate an Ecore model from Rational Rose (.mdl file), which is the approach taken in this paper to construct the Object-Z metamodel. Alternatively, we could create an Ecore model using an EMF supporting tool such as Omondo EclipseUML [6] which is a visual modelling tool that allows users to visually create and edit both UML and Ecore models, or from XML schema and other inputs.

Figure 7 shows the editor generated by EMF from the Object-Z metamodel presented in Section 2.1. When we click on the right button at the top of the tree editor, we can see all the model elements definable at the Object-Z specification level such as classes and data types. To create an Object-Z class instance, we simply choose Class from the list and fill in properties such as name in the property window. Once an instance of an Object-Z class is created, we can define attributes and operations using the editor. Once an Object-Z instance specification is created, EMF will generate an output in XMI that is an input to the Tefkat transformation engine. The example Object-Z model created using the editor in Figure 7 specifies a key system containing four classes resulting in the KeySystem.oz file. To view the Object-Z model in its concrete syntax (see Figure 8), we need to map the abstract syntax to a concrete syntax such as [21]. This work is under investigation.

[2] In this work, we use the UML2.ecore file supplied by the UML2 project [22].

3.2.2 Tefkat Transformation Engine

Tefkat is DSTC's prototype model transformation engine [2]. It is built on EMF, in that it works with Ecore models and corresponding XMI instances, and implements the DSTC's transformation language [3].



```
KeySystem.oz
Resource Set
platform:/resource/Source/KeySystem.oz
  OZ Specification
    Basic Type Integer
    Basic Type Boolean
    OZ Class Key
      Relationship Attribute rooms
      Pure Attribute number
      OZ Operation accessDenied
        OZ Parameter rm
        Operation Predicate
      OZ Operation accessGranted
        OZ Parameter rm
        Operation Predicate
    OZ Class MasterKey
    OZ Class Room
    OZ Class SingleRoomKey
      Relationship Attribute room
      Invariant
Selection | Parent | List | Tree | Table | TableTree
```

Properties

Property	Value
Class Type	OZ Class Room
Is Containment	true
Is Ordered	false
Is Static	false
Lower	1
Name	room
Type	OZ Class Room
Upper	1
Visibility Kind	Visible

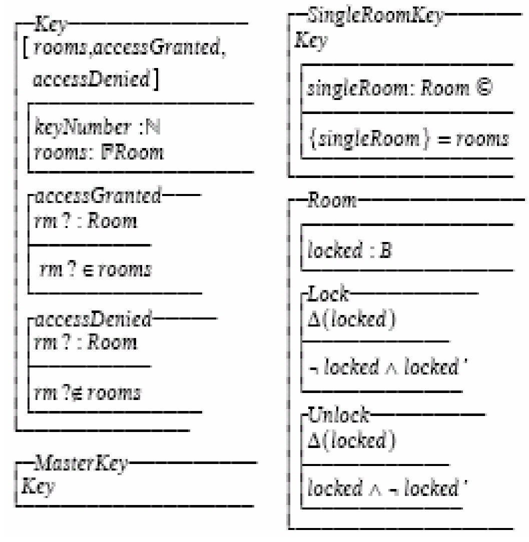


Fig. 7. Object-Z editor generated by EMF **Fig. 8.** Object-Z model in its concrete syntax

The user creates a Tefkat project containing the required Object-Z and UML metamodels, Object-Z source model, transformation rules file and possibly a tracking model. Tefkat includes a configuration editor which is used to create a configuration file specifying the locations of the required files. The transformation is set to run automatically each time the project is updated, creating a target UML instance. Tefkat also provides a textual editor for transformation rule files (.qvt), which highlights syntax and dynamically reports any syntax errors in the Eclipse Problems View.

4 Transformation Rules from Object-Z to UML

In this section, we describe how to convert Object-Z constructs to UML constructs using the DSTC transformation language and its Transformation Engine. The mapping is based on the metamodels of Object-Z and UML, and the tracking model defined in previous sections. Currently the DSTC's transformation language does not support bi-directional transformations [3]. For this reason, in this paper we define a set of transformation rules from Object-Z to UML, but the rules will be readily redefined when the transformation language supports the bi-directional feature.

4.1 Transformation Rule for Object-Z Classes

Semantically, an Object-Z class represents a set of objects of that class. This semantics is the same as that of a UML class, so we convert an Object-Z class to a UML class. The OZClass2UMLClass rule implements this mapping by creating a UML class for each OZClass. Line 5 declares the rule name and variables to be used in the rule. Line 7 introduces a WHERE...LINKS statement. This is the way in which the tracking relationship created in the OZSpec2UMLModel rule presented in Section 3.1 can be queried in order to find the correct UML model into which to place the created UML Class. Lines 7 and 8 effectively find the Object-Z specification that is containing the source Object-Z class (ozc.owner), and then look up and store the corresponding UML model (umlm) for use later in the rule. Line 9 creates the target UML class, while Line 10 introduces a SET statement, which is used to set the attributes and references of created target elements. In this case, the UML class name is set to the same name as the Object-Z class, and the UML class is added to the UML model. Note that umlc is being added to the ownedMember collection of the UML model, umlm[3]. Line 11 preserves the tracking relationship, also storing the corresponding Object-Z specification and UML model as these will be used in other rules.

```
5     RULE    OZClass2UMLClass(ozs, umlm, ozc, umlc)
6     FORALL  OZClass ozc
7     WHERE   OZSpecToUMLModel LINKS ozspec = ozs, umlmodel = umlm
8     AND     ozc.owner = ozs
9     MAKE    Class umlc
10    SET     umlc.name = ozc.name, umlm.ownedMember = umlc
11    LINKING OZClassToUMLClass WITH  ozclass = ozc, umlclass = umlc,
              ozspec = ozs, umlmodel = umlm;
```

4.2 Transformation Rule for Object-Z Operations

Each Object-Z operation is converted to a UML operation. The OZOperation2UMLOperation rule implements this mapping. This rule has a similar structure to OZClass2UMLClass, except that a UML Operation is created for each OZOperation and placed inside the correct UML Class (Line 19). However, the rule is different in that it demonstrates the use of two patterns in Lines 14 and 16. Many rules presented in this paper need to find the corresponding UML class for an Object-Z class. The lookupClass pattern on Line 21 simplifies this by defining the common WHERE...LINKS statement as a pattern so that it can be used in many rules. Also the pattern convertVisibility in Line 24 matches visibilities in both languages and it is used in the rules presented in this paper[4].

```
12    RULE    OZOperation2UMLOperation(ozc, umlc, ozo, umlo, vis)
13    FORALL  OZOperation ozo
14    WHERE   lookupClass(ozc, umlc)
15    AND     ozo.owner = ozc
16    AND     convertVisibility(ozo.visibilityKind, vis)
17    MAKE    Operation umlo
18    SET     umlo.name = ozo.name, umlc.ownedOperation = umlo,
              umlo.visibility = vis
```

[3] In the DSTC transformation language the syntax is the same for setting the value of a single-valued attribute or reference as it is for adding an element to a collection.

[4] Due to the page limits, we omit details of some pattern definitions.

```
19   LINKING OZOpToUMLOp WITH ozop = ozo, umlop = umlo,
             ozclass = ozc, umlclass = umlc;
20   // look up correct UML classes
21   PATTERN lookupClass(ozc, umlc)
22   WHERE   OZClassToUMLClass LINKS ozclass = ozc, umlclass = umlc;
23   // convert Object-Z visibility kinds into UML visibility kinds
24   PATTERN convertVisibility(ozvis, umlvis) …;
```

4.3 Transformation Rule for Object-Z Operation Parameters

Prior to describing the transformation of parameters, we explain how to convert data types. While types of attributes and parameters in UML are a language-dependent specification of the implementation types, those in Object-Z are language-independent specification types. For this reason, we define a rule to match only those types that are common in both languages such as Integer and Boolean (see the rule OZBasicType2UMLPrimitiveType) and we do not define a specific rule for other data types in Object-Z.

```
25   RULE    OZBasicType2UMLPrimitiveType(bt, pt, ozs, umlm)
26   FORALL  BasicType bt
27   WHERE   OZSpecToUMLModel LINKS ozspec = ozs, umlmodel = umlm
28   AND     bt.owner = ozs
29   MAKE    PrimitiveType pt
30   SET     pt.name = bt.name, umlm.ownedMember = pt
31   LINKING BasicTypeToPrimitiveType WITH basic = bt, prim = pt;
```

Due to the differences in data types in both languages, we apply different rules for parameters with different types. OZBasicParam2UMLParam is the rule to map the parameters of Object-Z operations with basic types to UML operation parameters with primitive types. Again, a WHERE...LINKS statement is used to find the correct UML operation for the created UML parameter in Line 34 and 35; to check the type (using the pattern isBasicType in Line 36); to find the correct matching UML type (using the pattern lookupBasicType in Line 37); to find the correct matching UML parameter kind (using the pattern convertParamKind in Line 38). Parameters in both languages have several equivalent properties that are mapped including name, isOrdered and upper and lower multiplicity values. The LiteralInteger and LiteralUnlimitedNatural classes must be used to set the upper and lower multiplicity values of the UML parameter because they are the types of the lowerValue and upperValue references respectively in the MultiplicityElement class of the UML2 metamodel (UML2.ecore).

```
32   RULE    OZBasicParam2UMLParam(ozo, umlo, ozp, umlp, int, nat,
             pkind, umltype)
33   FORALL  OZParameter ozp
34   WHERE   OZOpToUMLOp LINKS ozop = ozo, umlop = umlo
35   AND     ozp.owner = ozo
36   AND     isBasicType(ozp.type)
37   AND     lookupBasicType(ozp.type, umltype)
38   AND     convertParamKind(ozp.parameterKind, pkind)
39   MAKE    Parameter umlp, LiteralInteger int,
             LiteralUnlimitedNatural nat
40   SET     umlp.name = ozp.name, umlp.isOrdered = ozp.isOrdered,
             umlo.ownedParameter = umlp, int.value = ozp.lower,
             nat.value = ozp.upper, umlp.lowerValue = int,
             umlp.upperValue = nat, umlp.direction = pkind,
             umlp.type = umltype;
41   // convert Object-Z parameter kinds into UML parameter kinds
42   PATTERN convertParamKind(ozparkind, umlparkind)…;
43   // match OZ basic types with UML primitive types
44   PATTERN lookupBasicType(oztype, umltype)
```

```
45   WHERE   BasicTypeToPrimitiveType LINKS basic = oztype,
             prim = umltype;
46   // check Basic types
47   PATTERN isBasicType(oztype)
48   FORALL  BasicType oztype;
```

Rules for transforming parameters with other types are very similar to this rule except for the mapping of types. We omit these rules due to the page limits.

4.4 Transformation Rule for Object-Z Pure Attributes

Object-Z pure attributes (attributes with types that are not class types) are converted to a UML attribute. The OZBasicPureAttr2UMLProperty rule implements the mapping of pure attributes with basic types, setting the name, isStatic, isOrdered, lowerValue and upperValue properties of the created UML Property appropriately. Again a WHERE...LINKS statement is used to find the corresponding UML class for an Object-Z class in Line 51; to check the type in Line 52; to find the correct matching UML type in Line54; to find the correct matching UML visibility kind in Line 55.

```
49   RULE    OZBasicPureAttr2UMLProperty(ozc, umlc, oza, umlp, int,
             nat, vis, umltype)
50   FORALL  PureAttribute oza
51   WHERE   lookupClass(ozc, umlc)
52   AND     isBasicType(oza.type)
53   AND     oza.owner = ozc
54   AND     lookupBasicType(oza.type, umltype)
55   AND     convertVisibility(oza.visibilityKind, vis)
56   MAKE    Property umlp, LiteralInteger int,
             LiteralUnlimitedNatural nat
57   SET     umlp.name = oza.name, umlc.ownedAttribute = umlp,
             int.value = oza.lower, nat.value = oza.upper,
             umlp.isStatic    =    oza.isStatic,    umlp.isOrdered    =
             oza.isOrdered, umlp.lowerValue = int, umlp.upperValue =
             nat, umlp.visibility = vis, umlp.type = umltype;
```

4.5 Transformation Rule for Relationship Attributes

Object-Z RelationshipAttribute defines relationships between objects. This semantics of relationship attributes in Object-Z maps to that of associations in UML. The OZConRelAttr2UMLAssoc rule implements the mapping of relationship attributes with a containment property to UML Associations. Lines 60 - 62 are required to find the corresponding UML classes for the Object-Z class owning the relationship attribute and also the Object-Z class that is the type of that attribute (oza.classType). Line 63 matches the visibility, and Lines 64 and 65 match the containment property. An Association and two Properties are created as a result of the transformation rule. When an Object-Z class has a relationship attribute, the attribute is navigable by the owning class. In this rule, the nav property represents the navigable end of the association, while the non property is the non-navigable end. The SET statement in this rule accomplishes the following:

1. The name of the navigable end of the association is set to be the same as the name of the relationship attribute.
2. The type of the navigable end is set to the UML class corresponding to the Object-Z class that is the type of the relationship attribute.
3. The isStatic, isOrdered, lowerValue and upperValue properties of the navigable end are matched with the corresponding properties of the relationship attribute.

4. The type of the non-navigable end is set to the UML class corresponding to the Object-Z class that is the owner of the relationship attribute.
5. The navigable end property is added to the attributes of the UML class corresponding to the Object-Z class that is the owner of the relationship attribute. This is done because in the UML2.0 [19], a navigable end property of an association is also an attribute of the associated UML class.
6. In UML an Association has at least two memberEnd properties representing the ends of the association, and non-navigable ends are owned by the Association. This is accomplished at the end of Line 67 as both association ends are added to the memberEnd collection of the Association and the ownedEnd is set to the non-navigable end.
7. Finally, the new association is placed in the UML model by adding it to the ownedMember collection.

```
58   RULE    OZContRelAttr2UMLAssoc(ozc, umlc, umlm, oza, umla, non,
     nav, umlt, int, nat, vis, agg)
59   FORALL  RelationshipAttribute oza
60   WHERE   lookupClass(ozc, umlc)
61   AND     oza.owner = ozc
62   AND     OZClassToUMLClass LINKS ozclass = oza.classType,
             umlclass = umlt, umlmodel = umlm
63   AND     convertVisibility(oza.visibilityKind, vis)
64   AND     isContainment(oza)
65   AND     convertContainment(oza.isContainment, agg)
66   MAKE    Association umla, Property non, Property nav,
             LiteralInteger int1, LiteralUnlimitedNatural nat1,
             LiteralInteger int2, LiteralUnlimitedNatural nat2
67   SET     nav.name = oza.name, nav.type = umlt,
             nav.isStatic    =    oza.isStatic,    nav.isOrdered    =
             oza.isOrdered,   int1.value   =   oza.lower,   nat1.value   =
             oza.upper, nav.lowerValue = int1, nav.upperValue = nat1,
             nav.visibility = vis, non.aggregation = agg,
             int2.value = 0, nat2.value = 1,
             non.lowerValue = int2, non.upperValue = nat2,
             non.type = umlc, umlc.ownedAttribute = nav,
             umla.ownedEnd = non, umla.memberEnd = nav,
             umla.memberEnd = non, umlm.ownedMember = umla;
68   // check containment property
69   PATTERN isContainment(ozattr)
70   WHERE   ozattr.isContainment = true;
```

The rule for mapping relationship attributes with a non-containment property is basically the same as this rule except for the mapping of the containment property.

4.6 Transformation Rule for Inheritance

In Object-Z, inheritance is a mechanism to incrementally extend an Object-Z model. Sub-classes inherit all features defined in its super classes. We convert Object-Z inheritance to generalization in UML. The OZInherit2UMLGeneral rule achieves this mapping. For each pair of Object-Z classes where one is a superclass of the other, a UML Generalization is created. The lookupClass pattern is used twice to find the corresponding UML classes in Line 73 and 74. In the UML2.0 [19], a Generalization has specific and general references to store the subclass and superclass respectively, and the Generalization itself is owned by the subclass. These values are set appropriately in the SET statement. Since inheritance in Object-Z does not support subtyping automatically, we leave the isSubstitutable property of the generalization undefined.

```
RULE    OZInherit2UMLGeneral(ozc, umlc, ozg, umlg, umlgen)
FORALL  OZClass ozc, OZClass ozg
WHERE   lookupClass(ozc, umlc)
AND     lookupClass(ozg, umlg)
AND     ozc.general = ozg
MAKE    Generalization umlgen
SET     umlgen.specific = umlc, umlc.generalization = umlgen,
        umlc.generalization.general = umlg;
```

4.7 Transformation Example

Figure 9 shows the target UML model generated from the example source Object-Z model presented in Figure 8 according to the transformation rules defined in this section. The actual output (KeySystem.uml2) is in XMI but we visualize it using a UML class diagram.

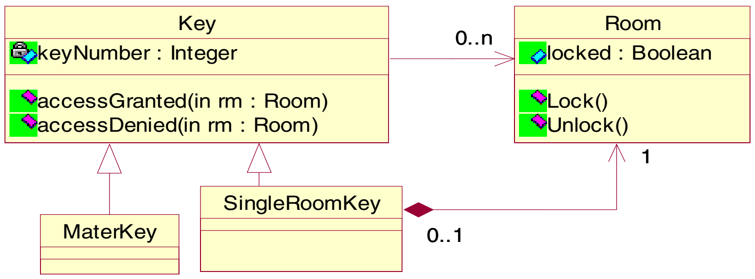


Fig. 9. A target UML model

5 Conclusion and Future Work

This paper has presented an MDA approach to integrating formal and informal modeling languages within the Eclipse tool integration environment. Using the MDA model transformation approach, we define a metamodel of Object-Z using the MOF. Given the metamodels of UML and Object-Z, we then define transformation rules using a transformation language, the DSTC's Transformation Language.

The metamodel-based MDA transformation framework allows us to define transformation rules precisely and explicitly, which is essential to be able to know the semantic basis of the transformation, to check the completeness and consistency of the transformation, and to provide the traceability of notations. It also allows us to have a reusable transformation that can be applied to any models in the two languages. Finally we achieve an automatic transformation using existing tools supporting MDA. In addition, the integrated approach with formal and informal techniques incorporates an effective V&V mechanism into the MDA and it supports model evolution that is concerned with correcting errors in the model.

The transformation rules presented in this paper are from Object-Z to UML. When the transformation language supports multi-directional transformation, the rules will be refined accordingly to support the bi-directional transformation between the two languages. Mapping the abstract syntax of Object-Z to its concrete syntax and converting Object-Z expressions to OCL expressions are under investigation.

Acknowledgements

This research is funded by an Australian Research Council Discovery Grant: DP0451830. We wish to thank Keith Duddy, Michael Lawley and other DSTC staff for their assistance with their transformation language [3] and their Tefkat tool [2].

References

1. S. Dascalu and P. Hitchcock, An approach to integrating semi-formal and formal notations in software specification, ACM symposium on Applied computing, pp. 1014–1020, 2002.
2. DSTC, Tefkat: The EMF transformation engine. http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/index.html
3. DSTC Transformation Language, MOF query/views/ transformations: Second revised submission, 2004. http://www.omg.org/docs/ad/04-01-06.pdf
4. R. Duke and G. Rose, Formal Object-Oriented Specification Using Object-Z, Macmillan, 2000.
5. Eclipse Foundation. http://www.eclipse.org/
6. EclipseUML, Omondo http://www.eclipsedownload.com/
7. EMF, The eclipse modeling framework. http://download.eclipse.org/tools/emf/scripts/docs.php?doc=references/overview/EMF.html
8. R. France, J. Wu, M. M. Larrondo-Petrie, and J.-M. Bruel, A Tale of Two Case Studies: Using Integrated Methods to Support Rigorous Requirements Specification, Proc. BCS FACS Methods Integration Workshop, 1996.
9. S-K. Kim and D. Carrington, A Formal Mapping between UML Models and Object-Z Specifications, ZB2000, LNCS 1878, pp. 2-21, 2000.
10. S-K. Kim, A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques, PhD Thesis, ITEE, The University of Queensland, 2002.
11. R. Laleau and F. Polack. Coming and going from UML to B: A proposal to support traceability in rigorous IS development. Proc. ZB'2002, LNCS 2272, pp. 517–534, 2002.
12. K. Lano, D. Clark and K. Androutsopoulos, UML to B: Formal Verification of Object-Oriented Models, Proc. IFM'04, LNCS 2999, pp. 187 - 206 2004.
13. J. Lilius and I. P. Paltor, Formalizing UML state machines for model checking, Proc. UML'99, LNCS 1723, pp. 430-445, 1999.
14. W. McUmber and B. Cheng. A General Framework for Formalizing UML with Formal Languages. in IEEE Conference on Software Engineering, pp. 433–442, 2001.
15. M. Y. Ng and M. Butler. Tool Support for Visualizing CSP in UML. Proc. ICFEM'02, LNCS 2495, pp. 287–298. 2002.
16. OMG, Meta Object Facility (MOF),1.4, OMG Document ad/02-04-03, 2002.
17. OMG, MOF 2.0 Query/Views/Transformations RFP, OMG Document ad/02-04-10, 2002.
18. OMG, MDA Guide Version 1.0.1, 2003. http://www.omg.org/docs/omg/03-06-01.pdf
19. OMG, UML 2.0 Superstructure Specification, OMG Document ptc/03-08-02. http://www.omg.org/docs/ptc/03-08-02.pdf, 2003.
20. S. Sendall and W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software, pp. 42-45, Sep/Oct 2003.

21. J. Sun, J. S. Dong, J. Liu, and H. Wang. Z family on the web with their UML photos. http://nt-appn.comp.nus.edu.sg/fm/zml/
22. UML2, The eclipse UML2 project. http://www.eclipse.org/uml2/
23. R. Wieringa, E. Dubois, and S. Huyts. Integrating Semi-formal and Formal Requirements. in Advanced Information Systems Engineering, LNCS 1250, pp. 19-32, 1997.