NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Specifying Coupling and Cohesion Metrics using OCL and Alloy

Jacqueline A. McQuillan
and
James F. Power

Department of Computer Science,
National University of Ireland, Maynooth
Co. Kildare, Ireland

Technical Report
NUIM-CS-TR-2008-02

Specifying Coupling and Cohesion Metrics using OCL and Alloy

Jacqueline A. McQuillan[1] and James F. Power
Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare
Ireland

{jmcq, jpower}@cs.nuim.ie

Abstract

This report presents a MOF-compliant metamodel for calculating software metrics and demonstrates how it is used to generate a metrics tool that calculates coupling and cohesion metrics. We also describe a systematic approach to the analysis of MOF-compliant metamodels and illustrate the approach using the presented metamodel. In this approach, we express the metamodel using UML and OCL and harness existing automated tools in a framework that generates a Java implementation and an Alloy specification of the metamodel, and use this both to examine the metamodel constraints, and to generate instantiations of the metamodel. Moreover, we describe how the approach can be used to generate test data for any software based on a MOF-compliant metamodel. We extend our framework to support this approach and use it to generate a test suite for the metrics calculation tool that is based on our metamodel.

Key words: object-oriented software metrics, coupling, cohesion, metamodels, model-based testing, OCL, Alloy

1 Introduction

Software metrics are important in many areas of software engineering, for example assessing software quality or estimating the cost and effort of developing software. Many metrics have been proposed and new metrics continue to appear in the literature regularly [9]. Many of these metrics are incomplete, ambiguous and open to a variety of different interpretations [3]. This makes it difficult to create general metric tools and everytime a new metric is defined the tools need to be updated with the new metric [16]. Furthermore, many of these metrics are applicable to a number of different models of a software system. In order to provide assurance that the same concepts are being measured from these different models we need a way to specify the metrics in a generic way, independent of the particular model.

[1]To whom correspondence should be addressed.

Like Mens and Lanza, we believe that these issues are best addressed using a *language-independent, metrics-specific metamodel* [16]. However, they do not consider coupling or cohesion metrics in their work. In this report we present a metamodel for calculating object-oriented software metrics which is based on existing frameworks for coupling and cohesion measurement [3, 4]. We use the metamodel to specify a set of existing coupling and cohesion metrics and use our existing Eclipse-based metrics framework [15] to automatically generate a tool to calculate these metrics.

Developing and working with metamodels can be difficult since they deal with abstract concepts. Therefore, it is important that we are able to perform analysis on metamodels and assess their correctness. By correctness, we mean that the metamodel specification is consistent and adequately describes what the user intends. Also, in order to ensure the correctness and quality of software applications that are based around metamodels, for example our metrics tool, we need to be able to test these applications. However, there is no direct way of automatically generating instantiations of a metamodel to use as test inputs for testing metamodel-based applications [7].

In this report we describe an approach to the analysis of Meta Object Facility (MOF)-compliant metamodels and apply it to our metrics specific metamodel. In our approach, we express the metamodel using the Unified Modelling Language (UML) and Object Constraint Language (OCL) [20], and harness existing automated tools in a framework that generates a Java implementation of our metamodel.

We also generate an Alloy [12] specification corresponding to the metamodel, and use this to examine the metamodel constraints and to generate sample instances of the metamodel. Our *reflective instantiator* takes these Alloy generated models and transforms them into instances of the Java implementation, thus harnessing Alloy's lightweight approach to generate a test suite for our metrics tool. We use this test suite to determine if the tool correctly computes metric values for the coupling and cohesion metrics. Finally, we evaluate the adequacy of the generated test suite in terms of traditional line and branch coverage criteria.

This report is organised as follows. Section 2 outlines some of the background information. Our metamodel is presented in Section 3 and section 4 describes how it is used to define a set of coupling and cohesion metrics. In Sections 5 and 6, we describe an approach to the analysis of MOF-compliant metamodels and illustrate the approach using our metamodel. In Section 7, we describe the generation of a test suite for the metamodel-based metrics tool. Section 8 presents a discussion of related work. Section 9 concludes the report.

2 Background

Models provide a representation of a real system and are increasingly important in software engineering, particularly in the Model Driven Engineering (MDE) approach [20]. Typically, we think of a model of a software system as being a design model, such as UML class or sequence diagrams, or an implementation model, such as an actual program. As the name suggests, a *meta*model is a model that is used to describe the structure of other models. One example of a metamodel specification is the *UML Superstructure Specification* from the Object Management Group (OMG) [19], which

defines models for each of the diagrams of the UML.

While a number of different formalisms may be used to describe metamodels, one of the most widely adopted standards is the Meta Object Facility (MOF), which is specified by the OMG [18]. The MOF provides a set of constructs for defining metamodels and is referred to as a *meta*metamodel. As well as facilitating comprehension, using a standard metamodelling formalism aides interoperability, through formats such as the XML Metadata Interchange (XMI), as well as automated tool generation.

2.1 Metamodels and metrics

Many software metrics have been proposed in the literature [6, 9]. For these software metric definitions to be usable, it is important that the definitions clearly specify what is to be counted. For example, when counting method calls, do we include calls to abstract methods, calls to/from inherited and overridden methods etc. ? Briand *et al.* have shown that even seemingly straightforward metric definitions are subject to a range of different interpretations [3, 4].

Several authors have considered the use of metamodels as a way to address the problem of ambiguous metric definitions. One example is the canonical presentation of coupling and cohesion metrics by Briand *et al.* which was effectively based on a metrics specific metamodel of an object-oriented system. Mens and Lanza [16] propose a language-independent metamodel for object-oriented metrics that is based on graphs. They use this to define a selection of generic object-oriented and higher order metrics but do not consider coupling or cohesion metrics.

Recent research has built upon this work by defining metrics as queries over metamodels. El-Wakil *et al.* propose the use of XQuery as a metric definition language to extract metric data from XMI documents, specifically UML designs [8]. Harmer and Wilkie, working from a relational schema, express metric definitions as SQL queries over this schema [21]. Baroni *et al.* propose using the OCL and the UML 1.3 metamodel to define UML-based metrics [2].

In our own work, we have extended the approach of Baroni *et al.* in a manner specifically designed to be reusable for other metamodels [15]. We used the *Dagstuhl Middle Metamodel* as a general programming metamodel, and defined several object-oriented metrics across this metamodel using OCL [14]. We have also defined similar metrics at the design level using the UML 2.0 metamodel [15].

However, each of these approaches is either metamodel-specific (e.g. the UML metamodel), or uses a *metrics specific* metamodel, with the associated difficulty of finding model instances to use as test data. The approach presented in this report addresses these issues by presenting a metrics-specific, language-independent metamodel and providing a means of generating suitable instances to use for testing any software based on this (or any MOF-compliant) metamodel. We acknowledge that even though the presented metamodel is language-independent it may need some work to apply to different styles of object-oriented language.

2.2 The Alloy language and analyser

Alloy is a formal specification language based on typed first-order relational logic [12]. It has been used primarily to explore abstract software models and to assist in finding and correcting flaws in these models. An Alloy specification is based around *signatures* and *formulas*. *Signatures* are used for defining the entities of the model and consist of a set of declarations that define the relations and operations of the entity. *Formulas* such as *facts*, *predicates* and *assertions* are used to specify constraints on the model.

A fully automatic tool, called the *Alloy Analyser* has been developed simultaneously with the Alloy language. This is a "model-finder" tool that uses a constraint solver to analyse models written in Alloy. There are two types of analysis offered by the tool, namely *simulation* and *checking*. *Simulation* involves finding model instances that meet the Alloy specification. *Checking* involves finding counterexamples to the specification. To make instance finding feasible, a user may specify a *scope* for the model under analysis. The *scope* puts a bound on how many instances of an entity may be observed in a model instance and thus limits the number of model instances to be examined.

3 A metamodel for object-oriented measurement

One requirement of our metamodel is that it is interoperable with the UML and Java metamodels and thus has been developed to conform to the MOF. This ensures that all three metamodels are specified using the same formalism, thus facilitating the translation of instances of the UML and Java metamodel to instances of the metamodel presented here. Moreover, our MOF-compliant metamodel is based on the coupling and cohesion measurement frameworks proposed by Briand *et al.*[3, 4]. It captures the basic structure of an object-oriented system at a level of abstraction that represents concepts and relationships required for coupling and cohesion measurement.

The metamodel is composed of a single package called MM (Metrics Metamodel) and the contents of this package are depicted in Figure 1. The figure shows the main classes involved in the metamodel, along with the important associations, necessary for distinguishing the different types of coupling and cohesion metrics. The basic details regarding the size of the metamodel are given in Table 1.

Element	**Number**
Class	8
Enumeration	2
Generalisation	3
Association	16
Attribute	4
Method	27

Table 1: Summary of metrics-specific metamodel. *This table gives a summary of the size of the metamodel in terms of the number of each type of element that appears in the metamodel.*

The description of our metamodel follows a similar format to that used by the OMG for the specification of metamodels. Each concept in the metamodel is described in its own subsection which is broken down into several different parts corresponding to different aspects of the concept. In situations where an aspect does not apply to the concept it is ommitted entirely from the description of the concept. Each concept is represented as a metaclass in the metamodel and described using the following

- The subsection heading gives the formal name of the concept.
- The *Description* aspect gives a brief, informal description of the meaning of the concept. Any direct generalisations of the concept are also detailed here.
- The *Attributes* aspect specifies each of the attributes that are defined for that metaclass. Each attribute is specified by its formal name and type. This is followed by an informal description of the meaning of the attribute.
- The *Associations* aspect lists all of the association ends owned by that metaclass. Again, each one is specified by its formal name, its type, and multiplicity and followed by an informal description of it's meaning.
- The *Constraints* aspect lists all of the constraints that define the well-formedness rules of the concept. Each constraint consists of an informal description and a formal constraint expressed in OCL.
- The *Operations* aspect contains a list of all the operations that belong to the metaclass. These include utility and query operations. In all cases each operation is specified using OCL. The utility operations are specified using the def keyword and the query operations are specified using the body keyword.

3.1 Attribute

3.1.1 Description

An attribute is an entity that describes a property of the class that it belongs to..

3.1.2 Associations

- **type:Type[1..1]** : specifies the type of the attribute
- **referenced_by:Method[0..*]** : specifies the set of methods that reference the attribute
- **att_implementing_class:Class[1..1]** : specifies the class in which the attribute is implemented
- **att_declaring_class:Class[0..*]** : specifies the set of classes in which the attribute is declared

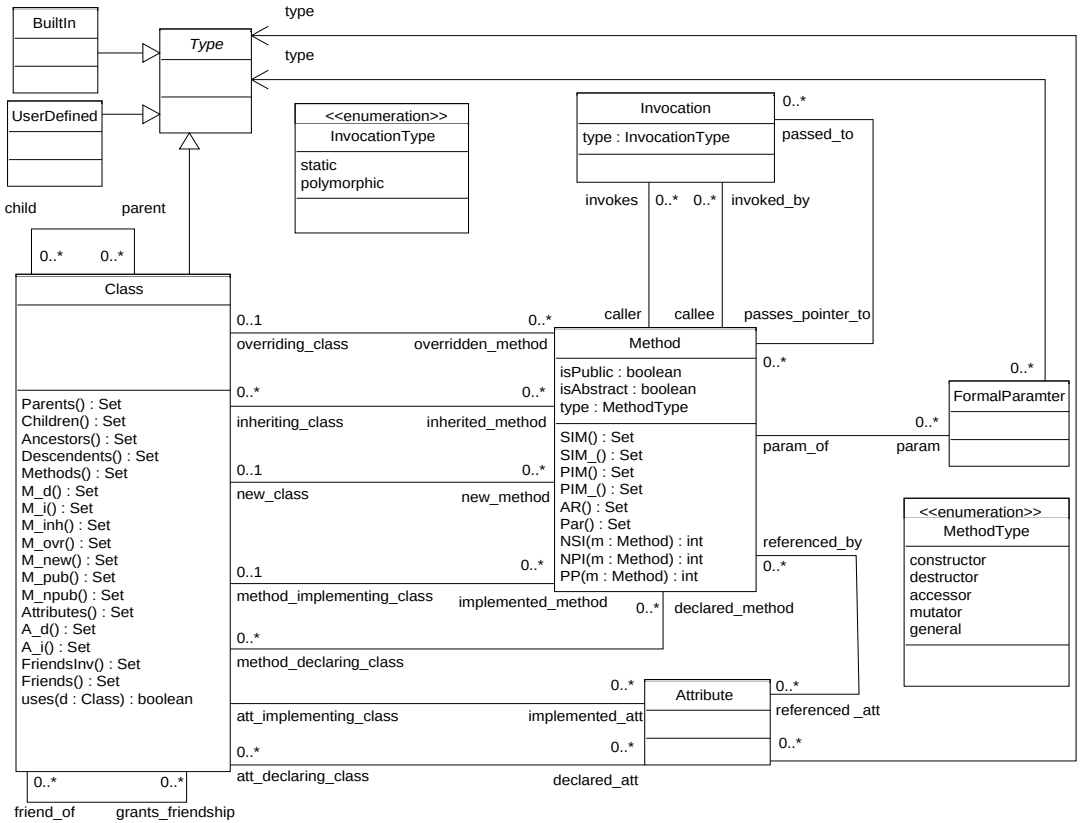

Figure 1: The metrics-specific metamodel. *This figure shows the main classes and relationships in our metamodel depicted as a UML class diagram.*

3.2 BuiltIn

3.2.1 Description

BuiltIn represents a basic type provided by the programming language (e.g., integer, real, character, string). It is a subclass of Type.

3.3 Class

3.3.1 Description

A class describes a set of entities that share the same properties and behaviour. It is a subclass of Type.

3.3.2 Associations

- **child:Class[0..*]** : specifies the set of immediate descendents of this class
- **parent:Class[0..*]** : specifies the set of immediate ancestors of this class
- **friend_of:Class[0..*]** : specifies the set of classes that are granted access to the non-public elements of this class
- **grants_friendship:Class[0..*]** : specifies the set of classes to which this class has access to the non-public elements of
- **overridden_method:Method[0..*]** : specifies the set of methods that are overridden by this class
- **inherited_method:Method[0..*]** : specifies the set of methods that are inherited by this class
- **new_method:Method[0..*]** : specifies the set of methods that are created as new in this class
- **declared_method:Method[0..*]** : specifies the set of methods that are declared in this class
- **implemented_method:Method[0..*]** : specifies the set of methods that are implemented in this class
- **declared_att:Attribute[0..*]** : specifies the set of attributes that are declared in this class
- **implemented_att:Attribute[0..*]** : specifies the set of attributes that are implemented in the class

3.3.3 Constraints

- A class may not directly or indirectly inherit from itself

```
not self.Ancestors()->includes(self)
```

- A class may not directly be a friend of itself or grant friendship to itself

```
not self.friend_of->includes(self)
```

- The set of declared attributes of a class must equal all the implemented attributes of that classes ancestors

```
self.declared_att = self.Ancestors()
                    ->collect(i:Class|i.implemented_att)->asSet()
```

- The set of new methods, overridden methods and inherited methods of a class must be disjoint

```
self.new_method->intersection(self.overridden_method)
               ->intersection(inherited_method)->isEmpty()
```

- The set of implemented methods of a class must equal the set of non-abstract, overriding methods union the set of non-abstract new methods of the class

```
self.implemented_method =
self.new_method->union(self.overridden_method)
               ->select(m:Method | not m.isAbstract)
```

- The set of declared methods of a class must be equal to the set of new abstract methods union the set of inherited methods of the class

```
self.declared_method =
self.inherited_method->union(self.new_method
->select(m:Method|m.isAbstract))
```

- The sum of the inherited and overridden methods of a class must equal the number of methods of the parents

```
not self.parent->isEmpty() implies
    self.inherited_method->union(self.overridden_method)->size()
    = self.parent->collect(i:Class|i.Methods())->asSet()->size()
```

- The set of inherited methods of a class must be a subset of the new and overriding methods of that classes ancestors

```
not self.inherited_method->isEmpty() implies
(self.Ancestors()->collect(i:Class|i.new_method)->asSet()
                 ->union(self.Ancestors()
                 ->collect(j:Class|j.overridden_method)
                 ->asSet()))
                 ->includesAll(self.inherited_method)
```

- If a class has no parentss then it cannot have any overridden methods

```
self.parent->isEmpty() implies self.overridden_method->isEmpty()
```

3.3.4 Operations

- The query A_d() returns the set of declared attributes of the Class.

```
Class::A_d():Set(Attribute)
body: self.declared_att
```

- The query A_i() returns the set of implemented attributes of the Class.

```
Class::A_i():Set(Attribute)
body: self.implemented_att
```

- The utility operation all_parents() returns the set of all direct and indirect ancestors of the Class.

```
def: all_parents(S:Set(Class)):Set(Class)
=self.parent->union((self.parent - S)
            ->collect(i:Class|i.all_parents(S->including(self))))
            ->asSet()
```

- The query Ancestors() returns the set of all direct and indirect ancestors of the Class.

```
Class::Ancestors():Set(Class)
body: self.all_parents(Set)
```

- The query Attributes() returns the set of all attributes belonging to the Class.

```
Class::Attributes():Set(Attribute)
body: self.declared_att->union(self.implemented_att)
```

- The query Children() returns the set of all immediate descendents of the Class.

```
Class::Children():Set(Class)
body: self.child
```

- The utility operation all_children() returns the set of all direct and indirect descendents of the Class.

```
def: all_children(S:Set(Class)):Set(Class)
=self.child->union((self.child - S)
           ->collect(i:Class|i.all_children(S->including(self))))
           ->asSet()
```

- The query Descendents() returns the set ofall direct and indirect descendents of the Class.

```
Class::Descendents():Set(Class)
body: self.all_children(Set)
```

- The query Friends() returns the set of direct friends of the Class.

```
Class::Friends():Set(Class)
body: self.friend_of
```

- The query FriendsInv() returns the set of inverse friends of the Class.

```
Class::FriendsInv():Set(Class)
body: self.grants_friendship
```

- The query M_d() returns the set of declared methods of the Class.

```
Class::M_d():Set(Method)
body: self.declared_method
```

- The query M_i() returns the set of implemented methods of the Class.

```
Class::M_i():Set(Method)
body: self.implemented_method
```

- The query M_inh() returns the set of inherited methods of the Class.

```
Class::M_inh():Set(Method)
body: self.inherited_method
```

- The query M_ovr() returns the set of overridden methods of the Class.

```
Class::M_ovr():Set(Method)
body: self.overridden_method
```

- The query M_new() returns the set of new methods of the Class.

```
Class::M_new():Set(Method)
body: self.new_method
```

- The query M_pub() returns the set of public methods of the Class.

```
Class::M_pub():Set(Method)
body: self.Methods()->select(m:Method|m.isPublic)
```

- The query M_npub() returns the set of nonpublic methods of the Class.

```
Class::M_npub():Set(Method)
body: self.Methods()->select(m:Method|not m.isPublic)
```

- The query Methods() returns the set of all methods that belong to the Class.

```
context Class::Methods():Set(Method)
body: self.declared_method->union(self.implemented_method)
```

- The query Parents() returns the set of direct ancestors of the Class.

```
Class::Parents():Set(Class)
body: self.parent
```

- The query uses(d) returns true if the Class uses attributes or methods belonging to the Class d.

```
Class::uses(d:Class):Boolean
body: self.implemented_method->collect(m:Method|m.PIM())
                              ->intersection(d.implemented_method)
                              ->notEmpty()
      or
      self.implemented_method->collect(m:Method|m.referenced_att)
                               ->intersection(d.implemented_att)
                               ->notEmpty()
```

3.4 FormalParameter

3.4.1 Description

FormalParameter represents an argument that is used to pass information in and out of a Method.

3.4.2 Associations

- **type:Type[1..1]** : specifies the type of the parameter
- **param_of:Method[1..1]** : specifies the method that the parameter belongs to

3.5 Invocation

3.5.1 Description

An invocation represents a method call.

3.5.2 Attributes

- **type:InvocationType** - specifies the type of the invocation i.e. wether it is static or dynamic

3.5.3 Associations

- **caller:Method[1..1]** : specifies the method in which the method call appears
- **calee:Method[1..1]** : specifies the method that is being called

- **passes_pointer_to:Method[0..*]** : specifies the set of methods that are being passed as a pointer during the invocation

3.6 InvocationType

InvocationType is an enumeration type that specifies the literals for defining the type of a method invocation.

3.6.1 Description

InvocationType is an enumeration of the following literal values:

- **static** : Indicates that the method is invoked statically.
- **dynamic** : Indicates that the method is invoked dynamically.

3.7 Method

3.7.1 Description

A callable function belonging to a class.

3.7.2 Attributes

- **type:MethodType** - specifies the type of the method
- **isAbstract:Boolean** - indicates if the method is abstract or non-abstract
- **isPublic:Boolean** - indicates if the method is public or non-public

3.7.3 Associations

- **overriding_class:Class[0..1]** : specifies the class that overrides this method
- **inheriting_class:Class[0..*]** : specifies the set of classes that inherit this method
- **new_class:Class[0..1]** : specifies the class in which this method is first defined
- **method_declaring_class:Class[0..*]** : specifies the set of classes in which this method is declared
- **method_implementing_class:Class[0..1]** : specifies the class in which this method is implemented
- **param:Parameter[0..*]** : specifies the set of parameters of this method
- **referenced_att:Attribute[0..*]** : specifies the set of attributes referenced by this method
- **invokes:Invocation[0..*]** : specifies the set of invocations in which this method is the caller

- **invoked_by:Invocation[0..*]** : specifies the set of invocations in which this metod is the callee
- **passed_to:Invocation[0..*]** : specifies the set of invocations where this method is passed as a parameter

3.7.4 Constraints

- If a method references at least one attribute, then this method must be non-abstract

```
self.referenced_att->notEmpty() implies self.isAbstract = false
```

- If a method is abstract then it must not call any methods

```
self.isAbstract = true implies self.invokes->isEmpty()
```

- If a method has no implementing class then it must be abstract

```
self.method_implementing_class->isEmpty() implies
self.isAbstract = true
```

- If a method is abstract then it must have a new class

```
self.isAbstract = true implies not self.new_class->isEmpty()
```

- A method must have either an overriding class or a new class

```
not self.overriding_class->asSet()->isEmpty() implies
    self.new_class->asSet()->isEmpty()
and self.overriding_class->asSet()->isEmpty() implies
    not self.new_class->asSet()->isEmpty()
```

- If a method is a constructor or destructor then it must not be abstract

```
self.type = MethodType::constructor or
self.type = MethodType::destructor
implies self.isAbstract = false
```

3.7.5 Operations

- The utility operation stat_invoked() returns the methods statically invoked by the Method.

```
def: stat_invoked():Bag(Method)
= self.invokes
  ->select(i:Invocation|i.type=InvocationType::static)
  ->collect(j:Invocation | j.callee)
```

- The utility operation poly_invoked() returns the methods polymorphically invoked by the Method.

```
def: poly_invoked():Bag(Method)
= self.invokes
  ->select(i:Invocation|i.type=InvocationType::polymorphic)
  ->collect(j:Invocation | j.callee)
```

- The utility operation closureSIM() returns the methods directly and indirevtly statically invoked by the Method.

```
def: closureSIM(S:Set(Method)):Set(Method)
= self.SIM()->union((self.SIM()-S)
            ->collect(m:Method|m.closureSIM(S->including(self)))
            ->asSet())
```

- The utility operation closurePIM returns the methods directly and indeirectly-polymorphically invoked by the Method.

```
def: closurePIM(S:Set(Method)):Set(Method)
= self.PIM()->union((self.PIM()-S)
            ->collect(m:Method|m.closurePIM(S->including(self)))
            ->asSet())
```

- The query operation AR() returns the set of attributes refernced by the Method.

```
Method::AR():Set(Attribute)
body: self.referenced_att
```

- The query operation NPI(m) returns the number of polymorphic invocations of m by the Method.

```
Method::NPI(m:Method):Integer
body: self.poly_invoked()->count(m)
```

- The query operation NSI(m) returns the number of static invocations of m by the Method.

```
Method::NSI(m:Method):Integer
body: self.stat_invoked()->count(m)
```

- The query operation Par() returns the set of parameters of the Method.

```
Method::Par():Set(FormalParameter)
body: self.param
```

- The query operation PIM() returns the set of methods polymorphically invoked by the Method.

```
Method::PIM():Set(Method)
body: self.poly_invoked()->asSet()
```

- The query operation SIM() returns the set of methods statically invoked by the Method.

```
Method::SIM():Set(Method)
body: self.stat_invoked()->asSet()
```

- The query operation PIM_() returns the set of indirectly polymorphically invoked methods of the Method.

```
Method::PIM_():Set(Method)
body: self.closurePIM(Set)
```

- The query operation SIM_() returns the set of indirectly statically invoked methods of the Method.

```
Method::SIM_():Set(Method)
body: self.closureSIM(Set)
```

- The query operation PP(m) returns the number of invocations of the Method where a pointer to the Method m is passed to this Method.

```
Method::PP(m:Method):Integer
body: self.invoked_by
      ->select(i:Invocation|i.passes_pointer_to->includes(m))
      ->size()
```

3.8 MethodType

MethodType is an enumeration type that specifies the literals for defining the type of a Method.

3.8.1 Description

MethodType is an enumeration of the following literal values:

- **constructor:** Indicates that the method is a constructor i.e. a method that is called when an object is created
- **destructor:** Indicates that the method is a destructor i.e. a method that is called when an object is destroyed
- **accessor:** Indicates that the method provides access to the attributes of the class to which it belongs
- **mutator:** Indicates that the method is used to modify the attributes of the class to which it belongs
- **general:** Indicates that the method is a general method that does't fall into any of the above categories

3.9 Type

3.9.1 Description

A type defines a set of values. Type is an abstract metaclass.

3.10 UserDefined

3.10.1 Description

A user-defined type of global scope (e.g., records, enumerations). It is a subclass of Type.

4 Defining metrics using the metamodel

In this section we describe how we used our metamodel to define three sets of existing object-oriented software metrics. The three sets of metrics *CKMetrics*, *Cohesion* and *Coupling* were taken from [6, 3, 4], respectively. In total, we defined 42 types of metrics and these are summarised in Table 2.

In keeping with the approach outlined in [15], the metrics were defined as OCL queries over the metamodel. The metamodel was extended with a separate metrics package containing a single class called *Metrics*, and each set of metrics was defined as follows:

1. A class was created in the metrics package for the metric set; this class extends the *Metrics* class.
2. For each metric, an operation was declared in the class, parameterised by the appropriate metamodel elements.
3. The metrics were defined by expressing them as OCL queries using the OCL `body` expression.

As an example of a definition, Figure 2 presents the definition of the **number of children** (NOC) metric. Here, the definition is parameterised by a single `Class`, and the body of the definition returns the size of the set of all children of this class. The auxiliary operation `Children` defined in the metamodel traverses the elements and

Metric Set	Metrics from references [6, 3, 4]
CKMetrics	WMC, NOC, DIT
Cohesion	LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Co, NewCo, TCC, LCC, ICH
Coupling	RFC, RFC', CBO, CBO', DAC, DAC', MPC, COF, ICP, IH_ICP, NIH_ICP, OCAEC, FCAEC, DCAEC, ACMIC, OCMIC, IFCMIC, AMMIC, OMMIC, IFMMIC, FMMEC, DMMEC, OMMEC, FCMEC, DCMEC, OCMEC, IFCAIC, ACAIC, OCAIC

Table 2: Summary of implemented metrics. *This table lists all 42 metrics that were implemented using our metamodel.*

relationships in the metamodel to assemble this set. Full details of this and the other metric definitions are detailed in the Appendices of this report.

We have developed a measurement framework for the definition and calculation of software metrics based around an Eclipse plug-in [15]. This framework was used to define all the metrics shown in Table 2 and to automatically create a tool to calculate these metrics. The metrics calculation tool was created by transforming the OCL and UML corresponding to the metric sets to Java code. In brief, the tool computes the metric values for metamodel instances by invoking the Java methods corresponding to the OCL metric definitions. In Section 7, we report on how we evaluate the correctness of this metrics tool.

```
-- Returns a count of the immediate
-- descendents of the Class c
context CKMetrics::NOC(c:MM::Class):Real
body: c.Children()->size()

-- Returns the set of children
context Class::Children():Set(Class)
body: self.child
```

Figure 2: Definition of the *NOC* metric. *This OCL code defnes the NOC metrics from the CK metrics set, and is part of a larger defnition of the whole CK metric set which we have implemented using the metrics-specific metamodel.*

5 An overview of the approach

In this section we present an overview of an approach that can be used to both analyse a MOF-compliant metamodel and to automatically generate test data for software based on the metamodel.

An overview of the approach is depicted in Figure 3. In this figure, our system is delineated by a dashed red line. The inputs to the system are the metamodel and its constraints expressed as UML and OCL, and are shown on the left of the figure. The outputs of the system are shown on the bottom, and consist of a Java implementation of the metamodel and its associated OCL constraints and queries, along with a test suite based on the metamodel. These are linked through a coverage analysis, as described in Section 7.

Both Octopus[2] and Alloy are third-party tools used in our system. The UML2Alloy tool used here is a re-implementation of the same tool of Anastasakis *et al.* [1], but specialised for Octopus. The *Reflective Instantiator* tool was developed by us. The process is almost fully automated, with user intervention limited to providing the original UML/OCL description of the metamodel, and examining the generated Alloy specification. This is depicted by the stick-figure in green in Figure 3.

There are six main steps in this process:

[2] `http://www.klasse.nl/octopus`

Step 1: Expressing the metamodel in UML and OCL. The OMG specification for MOF does not define a textual or graphical representation for MOF [18]. However, there is a UML Profile that defines a bi-directional mapping between UML and MOF. The profile facilitates the creation of metamodels using UML and the viewing of MOF metamodels. Our approach uses this to express the metamodel in UML and OCL. For example, MOF classes map to UML classes, MOF attributes to UML attributes and vice versa. In addition, any semantic constraints on the MOF metamodel map directly to UML constraints.

It is important to note that the profile is based on MOF 1.3 and has some limitations such as lacking a definition for mapping Enumeration. We choose to map from MOF Enumerations to UML Enumerations. This process was not automated, the metamodel was depicted using a standard UML modelling tool. Octopus is used to check the OCL for correct syntax and use of metamodel elements.

Step 2: Generating a Java implementation of the metamodel. After the metamodel and its constraints are depicted using UML/OCL, Octopus is used to generate the corresponding Java classes. The Octopus tool generates Java classes for each UML class. All attributes and associations in the metamodel are created as fields in the appropriate classes. Finally, Octopus creates methods to check that all the constraints of the metamodel have not been violated along with a method to check multiplicities of the metamodel.

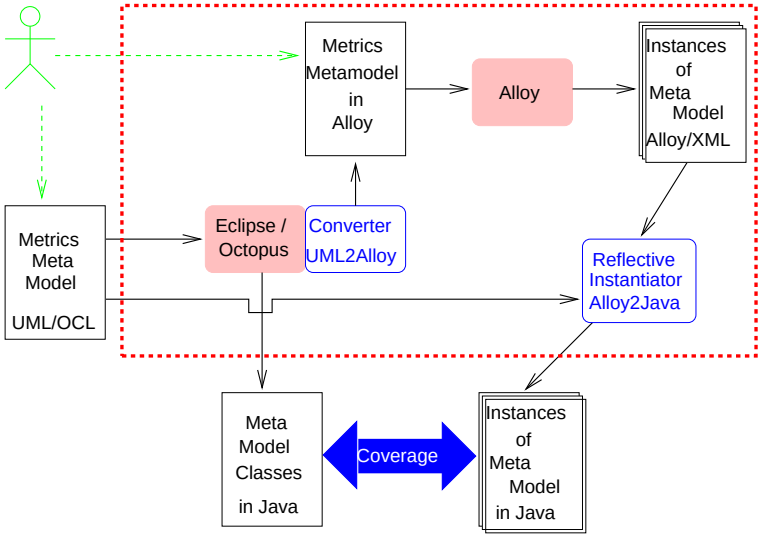

Figure 3: Overview of the approach to analysing the MOF-compliant metamodel. *The elements in the system are enclosed by the dashed red line. The input, shown on the left, are the metamodel and its constraints expressed as UML and OCL. The outputs, shown at the bottom, are the metamodel implementation and associated test instances in Java.*

Step 3: Transforming the metamodel to Alloy. We have created a tool to convert an Octopus UML/OCL metamodel to an Alloy specification. Since this tool mimics the *UML2Alloy* tool [1], we only briefly outline the transformation approach here.

UML classes are mapped to Alloy signatures. All attributes of a UML class are mapped to fields of the corresponding Alloy signature. UML Enumerators are mapped to abstract signatures in Alloy, with enumerator literals mapped to sub-signatures that extend the abstract signature of the enumeration. Basic UML data types are mapped to equivalent signatures from the Alloy library. The associations in the metamodel are also mapped to fields in the appropriate Alloy signatures. An additional fact is generated in the Alloy specification for bi-directional UML associations to show that the relations are symmetric.

Finally, any constraints on the metamodel in the form of OCL invariants are mapped directly to Alloy facts. At present, the OCL map does not cover the full language, and requires some user intervention for more difficult constructs.

Step 4: Analysis of the metamodel. The Alloy Analyser is used to analyse the Alloy model to detect flaws in the metamodel specification. For example, it can be used to generate random instances of the metamodel that conform to the well-formedness rules. If an instance cannot be found then there is an inconsistency in the metamodel specification. It is also possible to enumerate and explore all possible instances of the metamodel. This is useful to identify invalid instances i.e. instances that do not represent what the user intends their specification to represent.

Step 5: Generation of metamodel instances using Alloy. The role of Alloy in our system is twofold. First, it allows us to investigate the metamodel constraints to check for redundancies or errors (see Section 6). Second, it allows us to automatically generate valid metamodel instances. For this step we created a Java program to harness Alloy's model generation capabilities. This program reads an Alloy specification file and continually creates instances of the metamodel until all possible instances have been generated. Every metamodel instance produced during this step is output and stored in XML format for future use.

Step 6: Transformation of metamodel instances to Java objects. One of the central technical contributions of our system is the *Reflective Instantiator*, which transforms the XML versions of Alloy-generated models into instances of the Java implementation of our metamodel. The *Reflective Instantiator* parses the XML produced by Alloy and creates instances of our metamodel using the class files generated in Step 2. It does this using Java reflection, reading the class names from the XML files and creating instances of these classes. The fields of these classes are set by reading the fields from the XML and calling the appropriate set methods.

It is important to note that this process is not tied to any specific metamodel. Since the Alloy model and Java implementation of the metamodel are generated from the same MOF metamodel, Java reflection can make the link between them without having this information statically hard-coded. Therefore, this program is not specific to the metamodel under consideration and can be used for any metamodel.

6 Metamodel development and analysis

While the approach outlined in Section 5 will work for any MOF-compliant metamodel, our original intention was the specification and analysis of a metamodel for coupling and cohesion measurement. In this section we elucidate our approach using that metamodel.

6.1 Applying the approach

As described in Section 5, the first step of our approach is to express the metamodel in UML and OCL. As we were basing our metamodel on that of Briand *et al.*, we began by expressing the concepts described in [3, 4] as a class diagram and formalised any well-formedness rules that were expressed in natural language by Briand *et al.*. An example of such a rule is that *the set of all new, overriding and inherited methods of a class are disjoint*. We suspected that all these constraints were not sufficient to describe our metamodel and thus added 15 more constraints, resulting in a total of 27 well-formedness rules. Once we had formalised all of the rules in OCL, we used Octopus to statically check the OCL constraints and then translated the MOF-compliant metamodel and its well-formedness rules to Alloy.

An example of the translation of UML classes to Alloy is shown in Figure 4. This figure gives the Alloy specification for the $Class$ element of our metamodel which is defined in Alloy as a signature extending the $Type$ signature. The associations for a class are represented by fields, which we have shown here in four groups. These

```
sig Class extends Type
{
  /* Inheritance */
  parent: set Class,
  child: set Class,

  /* Friendship */
  grants_friendship: set Class,
  friend_of: set Class,

  /* Class - Attribute Relationships */
  declared_att: set Attribute,
  implemented_att: set Attribute,

  /* Class - Method Relationships */
  declared_method: set Method,
  implemented_method: set Method,
  new_method: set Method,
  overridden_method: set Method,
  inherited_method: set Method
}
```

Figure 4: Alloy signature for the `Class` element. This is a representation of the element `Class` in the Alloy specification lanhguage.

groups represent inheritance relationships, friendship relationships (for C++), and an association with the class' attributes and methods.

Furthermore, any constraints on the metamodel in the form of OCL invariants were mapped directly to Alloy facts. An example of such a constraint is depicted in Figure 5. This invariants states that *if a class does not have any parents then it cannot have any overridden methods* and maps to a fact in the Alloy specification.

```
-- OCL Specification:
inv noParentsThenNoOverriddenMethods :
  self.parent->isEmpty() implies self.overridden_method->isEmpty()

-- Alloy Specification:
fact noParentsThenNoOverriddenMethods
{
  all c:Class | c.parent = none implies c.overridden_method = none
}
```

Figure 5: An example of constraint on the metrics metamodel written in both OCL and Alloy. *This constraint states that if a class does not have any parents then it cannot have any overridden methods.*

6.2 Metamodel analysis

To perform the analysis, the Alloy Analyser was used to generate a random instance of the metamodel. The Analyser requires that a scope is specified for the model and then performs the analysis by exhaustively searching the state space for this scope. We specified a scope of 10 for all elements. The analyser searches for a model that contains at most 10 instances of each base class of the metamodel *and* conforms to the well-formedness rules of the metamodel. An instance was produced thus demonstrating that the well-formedness rules specified for the metamodel were consistent.

We used the Analyser to search for invalid metamodel instances. We specified a scope of 1 for the Alloy model and manually inspected the random instances produced by the Analyser. Each time an invalid instance was found, we added a constraint to prevent that instance from being generated. For example, we found a metamodel instance where a class could inherit from itself. On completion we had a total of 37 constraints.

Upon visual inspection of the 37 metamodel constraints, we suspected that a number of the constraints were superfluous. For each of these constraints, we converted it into an assertion about the metamodel and then used Alloy to check whether the assertion was valid. If the assertion produced a counterexample then we knew that the constraint was required. If a counterexample could not be found within a reasonable scope then it cannot be guaranteed that the constraint is redundant but it can increase our confidence that it is. Therefore, we assumed that the constraint was superfluous and omitted it from the specification. During this final analysis, 24 constraints were identified as potentially redundant and removed from the Alloy specification. We also found that a further 2 constraints were needed to prevent invalid metamodel instances,

thus giving us a total of 15 constraints in the Alloy version of the metamodel.

6.3 Discussion

This approach relies on Jackson's *small scope hypothesis*, which suggests that if a bug exists it will appear in *fairly small* models of a system [12]. So, it is possible our approach may not be applicable to larger metamodels. However, in such a situation it may be possible to apply the approach by partitioning and abstracting the metamodel into the parts that are related to the properties being analysed.

Moreover, we are fully aware that this process is not a completely formalised method for developing and analysing metamodels. However, we believe that this approach gives the developer a formal way of analysing and checking for any suspected deficiencies in their metamodel specification. By iteratively analysing and improving the metamodel, the developer becomes more confident in their specification.

Finally, it is important to note that this approach is not specific to a particular metamodel. It is generally applicable to any MOF-compliant metamodel. In fact, the approach is not restricted to metamodels but is applicable to any kind of model, for example a UML class diagram of a UML model.

7 Test suite generation

As described in Sections 3 and 5 we were able to automatically generate both an implementation of the metamodel and an implementation to calculate the specified metrics. In this section we describe the final step in integrating the use of Alloy with this code: the construction of a test suite for the automatically generated metrics tool. We use this test suite as input to the metrics tool and use a test oracle to determine whether or not the metric results produced by the tool are correct. The test oracle had to be constructed manually and therefore, required a test suite with the following properties:

1. Each test case should contain a relatively small number of elements.
2. The number of test cases in the test suite should also be relatively small.
3. The test suite should provide as much coverage of the implementation as possible.

Test Group	**Alloy Command**	**No. of Test Cases**
1	run show for exactly 1 Type, exactly 1 Attribute, exactly 1 Method, exactly 1 FormalParameter, exactly 1 Invocation	40
2	run show for 1	217
3	run show for exactly 1 *... all classes listed*	360
4	run show for exactly 2 Type, exactly 2 Attribute, exactly 2 Method, exactly 2 FormalParameter, exactly 2 Invocation	528,152

Table 3: Groups of test cases. *There were four main groups of test cases, generated by varying the settings for Alloys model generator. The number of test cases in each group is shown in the final column.*

7.1 Test case generation

Using our reflective instantiator described in Section 5 we were able to automate the generation of a set of test cases for the metrics calculation tool. As we required models with a relatively small number of elements we began by generating models using a small scope. Table 3 summarises the results of generating these test cases which are partitioned into four different groups:

Group 1 consisted of all possible instances with exactly one instance of each base class in our metamodel.

Group 2 is all possible instances where each base class is observed 0 or 1 times in a metamodel instance.

Group 3 is similar to group 1 except that we defined a scope of exactly 1 for all classes (not just base classes).

Group 4 again is similar to group 1 except that we allowed a scope of exactly 2 for all base classes.

7.2 Test cases and expected results

We added the two extra constraints to the original UML/OCL specification, and thus the generated Java implementation contained 39 constraints in total. All of the test cases summarised in Table 3 were used as input to our Reflective Instantiator. For each model, the Instantiator built the instantiation, ran the code to check each of the 39 OCL constraints, and then systematically tore down each model to test the element removal code. As each test model was built it was used as input to the metrics calculation tool and the values for all 42 metrics were recorded.

Since each generated constraint was checked for each test case, this provided further assurance that the reduced set of constraints used to generate the Alloy models was sufficient. Further, using such a large number of test cases demonstrates the robustness of the metric calculation tool and was used as a *smoke test* to ensure that the recorded values were within reasonable boundaries. Based on the scope used to generate each of the groups in Table 3 we computed the maximum and minimum values possible for each of the metrics. We then identified the models that produced metric values outside of these bounds. The results of this smoke test are discussed later in this section.

Our original intention was to generate a test suite with a relatively small number of test cases whose metric values could be calculated manually, serving as a test oracle for the generated metrics tool. However, since the number of test cases produced is in excess of 500,000, it is necessary to reduce this suite to a more manageable size. We decided to measure the coverage of the implementation in terms of traditional code coverage criteria and to reduce the number of test cases based on these criteria.

7.3 Coverage analysis

Cobertura[3] was used to measure the line and branch coverage of the metamodel implementation and the implementation corresponding to the metrics. Cobertura is a free Java tool that computes the percentage of code accessed by tests.

[3] `http://www.cobertura.sourceforge.net/`

Reason for exclusion	Line	Branch
Negative test cases	11%	1%
Field setters	6%	2%
Passed-as-Pointer Association	3%	4%
Total excluded	20%	7%

Table 4: Line/Branch coverage excluded from the coverage targets. *This table lists five kinds of code excluded from the coverage targets, along with the percentage of lines/branches for each kind.*

Test	Cum. Line Coverage			Cum. Branch Coverage		
Group	MM	Metrics	All	MM	Metrics	All
1	44%	68%	51%	55%	60%	57%
2	49%	68%	54%	62%	60%	61%
3	49%	68%	54%	62%	60%	61%
4	71%	99%	79%	91%	99%	93%

Table 5: A breakdown of the metamodel coverage for each of the test groups in Table 3. *The numbers presented for each group represent the cumulative coverage achieved, including the previous test groups.*

It was not possible to achieve full line and branch coverage of the implementation for several reasons, summarised in Table 4. Since our test suite only included positive test cases, code that involves catching exceptions when the invariants of the metamodel are violated was not fully covered. Some auxiliary routines, such as alternative set and get methods were not called in constructing the model. For simplicity, the part of the metamodel dealing with method pointers was not instantiated in Alloy, significantly reducing the number of models created. Thus, excluding these totals, from our target coverage gave a maximum possible coverage of 80% for line and 93% for branch coverage.

The results of the coverage analysis is summarised in Table 5 on a per-group basis. This table has one row for each of the test case groups described previously in Table 3. The data in each case represents the percentage coverage for each of the two coverage criteria. Each row describes the percentage coverage of the metamodel implementation (MM), the metrics implementation (Metrics) and the combined percentage coverage (All). Furthermore, each row represents *cumulative* coverage; for example, the line coverage value of 54% for group 2 includes the 51% line coverage achieved by group 1. As can be seen from Table 5, the smaller test suites exhibit relatively poor coverage.

7.4 Test oracle construction

In this subsection we consider the construction of a *reduced* test suite that achieves the maximum coverage criteria possible for use as a test oracle for the metrics tool.

A number of techniques exist that can reduce test suites based on various constraints. For example, Harrold *et al.* outline techniques for test suite reduction and prioritisation based on coverage criteria [11]. However, since our test cases were being

Test Case	Cum. Line Coverage			Cum. Branch Coverage		
	MM	Metrics	All	MM	Metrics	All
T1	43%	66%	50%	55%	59%	56%
T2	44%	68%	51%	55%	60%	57%
T3	44%	68%	51%	55%	60%	57%
T4	44%	68%	51%	56%	60%	57%
T5	48%	68%	54%	62%	60%	61%
T6	59%	68%	54%	62%	60%	61%
T7	63%	88%	71%	80%	88%	83%
T8	68%	89%	74%	87%	89%	87%
T9	68%	89%	74%	87%	89%	88%
T10	68%	89%	74%	87%	89%	88%
T11	69%	97%	77%	87%	98%	90%
T12	69%	97%	77%	87%	98%	90%
T13	69%	98%	77%	87%	99%	91%
T14	71%	99%	79%	91%	99%	93%

Table 6: The test cases in the reduced test suite. *This table lists the 11 test cases in the reduced suite, along with the cumulative coverage figures under each of the five coverage criteria.*

generated by Alloy roughly in order of size, a simpler approach was taken to test suite reduction:

1. As each test case is executed, the cumulative coverage of both criteria is recorded.
2. Any test case that causes an increase in any one of the two coverage figures is added to the reduced suite.
3. This process is continued until either the maximum coverage has been achieved for both criteria or until all test cases have been examined.

In general this process will not perform as well as that of Harrold *et al.*, but it is much simpler to implement. Applying this technique to the test cases, we generated a reduced test suite of 14 unique test cases. Table 7 lists the cumulative coverage data for each of these cases, labelled T1-T14. Three of these cases (T1-T3) originated from group 1, three (T4-T6) from group 2, and eight (T7-T14) from group 4.

The 14 test cases almost achieved the maximum coverage possible. By inspecting the output from the Cobertura tool we were able to identify 10 lines of code that had not been covered by the reduced test suite. We then used Alloy to generate a valid metamodel instance to cover this situation. This model was added to our test suite and increased the coverage to the maximum value possible of 80% for code coverage and 93% for branch coverage.

The 15 test cases were then used to manually create a test oracle for the metrics tool. All 42 metrics were calculated by hand and recorded for each of the 15 test cases. We compared these values with the actual values computed by the metrics tool. In the next subsection, we briefly discuss the results of this along with the results from the smoke test.

7.5 Discussion

Using the above procedure we uncovered 6 bugs in the metrics tool. Four of these were detected by the smoke test and 2 with the test oracle. For example, for certain cohesion metrics (e.g. LCOM1), an auxiliary operation was specified in OCL to compute the set of method pairs in a $Class$. It was discovered that each method pair was being counted twice and thus returning a metric value outside of the expected bounds for the metrics. This error was corrected at the OCL level. Further, we identified and fixed the remaining bugs and regenerated the metrics tool.

In summary, we were able to partition the types of errors we found into three categories. The first category are bugs that are a result of the metric definitions themselves. For, example when a metric has no provision for a division by 0. Second, are those introduced in the OCL where the definition has been incorrectly specified, for example a misplaced bracket in the OCL definition. Lastly, errors introduced by Octopus in transforming the UML/OCL to Java, for example incorrect casting of objects. Overall, our experience found this to be a relatively simple and effective way of increasing our confidence in the correctness of the automatically generated metrics tool.

8 Related work

The parallel between specification in Alloy and modelling in UML has been noted by Massoni *et al.* [13] and exploited by Anastasakis *et al.* [1]. Anastasakis *et al.* present a tool, *UML2Alloy*, that takes a UML class diagram, along with the associated OCL constraints, and translates this into an Alloy specification. The sample instances generated by the Alloy Analyser then correspond to object diagrams from the UML model. However, their tool does not provide any automated handling of the generated Alloy models.

Several other researchers have used Alloy to analyse and reason about metamodels. For instance, an alternative definition of the UML metamodel is presented in [17] and analysed using Alloy. In [22], Alloy is used to formalise and analyse the package merge concept of the UML 2.0 metamodel. These approaches are similar to ours in that they use Alloy to describe a *meta*model, as opposed to a *model* as with Anastasakis *et al.*. However, the main focus of this research to date has been on the analysis of the UML metamodel. Our work, is concerned with using Alloy to analyse a metamodel for object-oriented software measurement. Moreover, these approaches have no automated support for metamodelling or for handling the generated models.

Some work related to ours is that of Gogolla *et al.* [10] who describe an approach to the automatic generation of model instances (snapshots) from UML class diagrams. ASSL (A Snapshot Sequence Language) is used to specify properties of a required model instance. Using their approach they generate two types of model instances, those that are test cases and those that are validation cases. The test cases confirm that models with certain properties can be created from the specification. The validation cases are used to show that certain properties of a model are a consequence of existing properties of the model. However, this approach is not fully automated as it requires the creation of scripts for each model in order to generate instances.

A related problem is that of generating metamodel instances for use in testing model transformations. Brottier *et al.* use an approach that determines the part of the metamodel that is relevant to the model transformation, and then determines coverage criteria based on this part of the metamodel [5]. This criteria is then used to generate metamodel instances. However, OCL constraints, an important part of a metamodel, cannot be directly reflected, leading to an under-specification of model instances.

Finally, an approach to metamodel instance generation is presented by Ehrig *et al.* [7]. This approach involves the automatic creation of an instance-generating graph grammar for the given metamodel. They also describe how to translate restricted OCL constraints to graph constraints. The grammar and the graph constraints are then used to create metamodel instances. However this approach does not support attribute values, only supports limited OCL constraints and cannot be used to verify properties of the metamodel.

9 Concluding remarks

In this paper we presented an approach to analysing MOF-compliant metamodels. We also presented a metamodel for coupling and cohesion measurement based on the work of Briand *et al.* and described how we used our approach to construct and analyse the metamodel. The metamodel and well-formedness rules were expressed in UML and OCL and a Java implementation and Alloy specification of the metamodel were generated by third-party tools.

We used the Alloy specification to examine and validate the metamodel constraints, and to generate instantiations of the metamodel. We implemented a reflective instantiator to transform the automatically generated Alloy models into an instantiation of the Java implementation of the metamodel, generating a test suite for the metamodel-based metric calculation tool. finally, we evaluated the adequacy of the test suite using several coverage criteria.

We identify the principal contributions as:

- The **development and analysis** of a MOF-compliant metamodel for coupling and cohesion metrics, based on the work of Briand *et al.*, and the elimination of redundant constraints in that metamodel.
- The **automation** of the generation of metamodel instances from a UML/OCL specification that can be used as test data for metamodel-based software.
- A **coverage-based analysis** of the Alloy-generated test suite in terms of code coverage, thus "completing the circle" between lightweight formal methods and standard software testing techniques.

In future work, we plan to define a precise and complete scheme for transforming UML models and Java programs to instances of the metamodel presented in this report. Also, we believe the metamodel can be easily extended to other types of object-oriented metrics simply by expanding it with the new concepts required for the different types of metrics.

A Chidamber and Kemerer Metric Definitions

```
-- Returns a count of all the implemented methods of the Class c
context CKMetrics::WMC(c:MM::Class):Real
body: c.M_i()->size()

-- Returns a count of all the immediate descendents of the Class c
context CKMetrics::NOC(c:MM::Class):Real
body: c.Children()->size()

-- Computes the DIT for the Class c
context CKMetrics::DIT(c:MM::Class):Real
body: if c.Parents()->size() = 0 then --current Class c is root
         0
      else --DIT for Class c is maximum DIT value of its parents
         self.max(c.Parents()->collect(i:MM::Class|self.DIT(i)+1)
                              ->asSet())
      endif
```

B Coupling Definitions

```
def: ClassesAsTypes():Set(MM::Type)
= MM::Class.allInstances()->asSet()

def: CA(c:MM::Class, d:MM::Class):Real
= c.A_i()->select(a|a.type=d)->size()

def: CM(c:MM::Class, d:MM::Class):Real
= sumInts(c.M_new()->collect(m|m.Par()->select(a|a.type=d)->size()))

def: MM(c:MM::Class, d:MM::Class):Real
= sumReals(c.M_i()->collect(m|sumInts(d.M_new()->union(d.M_ovr())
                  ->collect(m1|m.NSI(m1) + m.PP(m1)))))

def: Others(c:MM::Class):Set(MM::Class)
= MM::Class.allInstances()-
  (c.Ancestors()->union(c.Descendents())->union(c.Friends())
                ->union(c.FriendsInv())->including(c))

-- Computes the CBO for the Class c
context Coupling::CBO(c:MM::Class):Real
body: MM::Class.allInstances()->excluding(c)
     ->select(d:MM::Class|c.uses(d) or d.uses(c))->size()

-- Computes the CBO' for the Class c
context Coupling::CBO_(c:MM::Class):Real
body: (MM::Class.allInstances()-(c.Ancestors()->including(c)))
       ->select(d:MM::Class|c.uses(d) or d.uses(c))->size()

-- Computes the RFC for the Class c
context Coupling::RFC(c:MM::Class):Real
body: c.Methods()->collect(m:MM::Method|m.PIM())
                 ->asSet()->union(c.Methods())->size()

-- Computes the RFC' for the Class c
context Coupling::RFC_(c:MM::Class):Real
body: c.Methods()->collect(m:MM::Method|m.PIM_())
                 ->asSet()->union(c.Methods())->size()

-- Computes the DAC for the Class c
context Coupling::DAC(c:MM::Class):Real
body: c.A_i()->select(a|self.ClassesAsTypes()
             ->includes(a.type))->size()

-- Computes the DAC' for the Class c
context Coupling::DAC_(c:MM::Class):Real
body: c.A_i()->collect(a|a.type)->asSet()
             ->select(t:MM::Type|self.ClassesAsTypes()
             ->includes(t))->size()

-- Computes the MPC for the Class c
context Coupling::MPC(c:MM::Class):Real
body: sumReals(c.M_i()
      ->collect(m:MM::Method|sumInts((m.SIM()-c.M_i())
      ->collect(m1:MM::Method|m.NSI(m1)))))
```

```
-- Computes the COF for the entire system
context Coupling::COF():Real
body: let numClasses:Real = MM::Class.allInstances()->size() in
      ((sumInts(MM::Class.allInstances()
                ->collect(c|(MM::Class.allInstances()-
      (c.Ancestors()->including(c)))->select(d|c.uses(d))->size())))
      / ((numClasses*numClasses) - numClasses -
      (2*(sumInts(MM::Class.allInstances()
                  ->collect(c|c.Descendents()->size()))))))

-- Computes the ICP for the Method m
context Coupling::ICP(c:MM::Class, m:MM::Method): Real
body: sumInts((m.PIM()-(c.M_new()->union(c.M_ovr())))
      ->collect(m1|(1 + m1.Par()->size())*m.NPI(m1)))

-- Computes the ICP for the Class c
context Coupling::ICP(c:MM::Class): Real
body: sumReals(c.M_i()->collect(m|self.ICP(c, m)))

-- Computes the ICP for the entire system
context Coupling::ICP(): Real
body: sumReals(MM::Class.allInstances()->collect(c|self.ICP(c)))

-- Computes the NIH_ICP for the Method m
context Coupling::NIH_ICP(c:MM::Class, m:MM::Method): Real
body: sumInts((m.PIM()->intersection( c.Ancestors()
                    ->collect(a|a.Methods())))
                    ->collect(m1|(1+m1.Par()->size())*m.NPI(m1)))

-- Computes the NIH_ICP for the Class c
context Coupling::NIH_ICP(c:MM::Class): Real
body: sumReals(c.M_i()->collect(m|self.NIH_ICP(c, m)))

-- Computes the NIH_ICP for the entire system
context Coupling::NIH_ICP(): Real
body: sumReals(MM::Class.allInstances()->collect(c|self.NIH_ICP(c)))

-- Computes the IH_ICP for the Method m
context Coupling::IH_ICP(c:MM::Class, m:MM::Method): Real
body: sumInts((m.PIM()->intersection((MM::Class.allInstances() -
     (c.Ancestors()->including(c)))->collect(a|a.Methods())))
     ->collect(m1|(1 + m1.Par()->size())*m.NPI(m1)))

-- Computes the IH_ICP for the Class c
context Coupling::IH_ICP(c:MM::Class): Real
body: sumReals(c.M_i()->collect(m|self.IH_ICP(c, m)))

-- Computes the IH_ICP for the entire system
context Coupling::IH_ICP(): Real
body: sumReals(MM::Class.allInstances()
               ->collect(c|self.IH_ICP(c)))

context Coupling::IFCAIC(c:MM::Class): Real
body: sumReals(c.FriendsInv()->collect(d|self.CA(c,d)))

context Coupling::ACAIC(c:MM::Class): Real
```

```
body: sumReals(c.Ancestors()->collect(d|self.CA(c,d)))

context Coupling::OCAIC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.Friends())
                              ->collect(d|self.CA(c,d)))

context Coupling::FCAEC(c:MM::Class): Real
body: sumReals(c.Friends()->collect(d|self.CA(d,c)))

context Coupling::DCAEC(c:MM::Class): Real
body: sumReals(c.Descendents()->collect(d|self.CA(d,c)))

context Coupling::OCAEC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.FriendsInv())
                              ->collect(d|self.CA(d,c)))

context Coupling::IFCMIC(c:MM::Class): Real
body: sumReals(c.FriendsInv()->collect(d|self.CM(c,d)))

context Coupling::ACMIC(c:MM::Class): Real
body: sumReals(c.Ancestors()->collect(d|self.CM(c,d)))

context Coupling::OCMIC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.Friends())
                              ->collect(d|self.CM(c,d)))

context Coupling::FCMEC(c:MM::Class): Real
body: sumReals(c.Friends()->collect(d|self.CM(d,c)))

context Coupling::DCMEC(c:MM::Class): Real
body: sumReals(c.Descendents()->collect(d|self.CM(d,c)))

context Coupling::OCMEC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.FriendsInv())
                              ->collect(d|self.CM(d,c)))

context Coupling::IFMMIC(c:MM::Class): Real
body: sumReals(c.FriendsInv()->collect(d|self.MM(c,d)))

context Coupling::AMMIC(c:MM::Class): Real
body: sumReals(c.Ancestors()->collect(d|self.MM(c,d)))

context Coupling::OMMIC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.Friends())
                              ->collect(d|self.MM(c,d)))

context Coupling::FMMEC(c:MM::Class): Real
body: sumReals(c.Friends()->collect(d|self.MM(d,c)))

context Coupling::DMMEC(c:MM::Class): Real
body: sumReals(c.Descendents()->collect(d|self.MM(d,c)))

context Coupling::OMMEC(c:MM::Class): Real
body: sumReals(self.Others(c)->union(c.FriendsInv())
                             ->collect(d|self.MM(d,c)))
```

C Cohesion Definitions

```
-- Returns a value for the common attribute usage measure
def: cau(m1:MM::Method, m2:MM::Method, c:MM::Class):Boolean
= (m1.SIM_()->including(m1)->collect(i:MM::Method|i.AR()))
            ->intersection(m2.SIM_()->including(m2)
            ->collect(j:MM::Method|j.AR()))->intersection(c.A_i())
            ->notEmpty()

-- The transitive closure of cau
def: cau_(m1:MM::Method, m2:MM::Method, c:MM::Class):Boolean
= (m1.SIM_()->including(m1)->collect(i:MM::Method|i.AR()))
            ->intersection(m2.SIM_()->including(m2)
            ->collect(j:MM::Method|j.AR()))->intersection(c.A_i())
            ->notEmpty()

-- Returns a set of sets of Methods, where each set represents
a connected component
def: constructConnectedComponents(v:Set(MM::Method),
     e:Set(TupleType(m1:MM::Method, m2:MM::Method)))
      :Set(Set(MM::Method))
= v->iterate(m:MM::Method;result2:Set(Set(MM::Method)) = Set |
  if not result2->flatten()->includes(m) then
     result2->including(self.getConnectedElements(m, e)
            ->including(m))
  else result2
  endif )

-- Returns the set of methods that along with m make a
single connected component
def: getConnectedElements(m:MM::Method,
     :Set(TupleType(m1:MM::Method, m2:MM::Method)))
     :Set(MM::Method)
= self.closureConnectedElements(m, e, Set)

-- Returns the set of methods that are transitively connected to m
def: closureConnectedElements(m:MM::Method,
     e:Set(TupleType(m1:MM::Method, m2:MM::Method)),
     S:Set(MM::Method))
      :Set(MM::Method)
= self.connectedElements(m,e)->union((self.connectedElements(m,e)-S)
  ->collect(i:MM::Method |
  self.closureConnectedElements(i, e, S->including(m)))->asSet())

--Returns the set of methods directly connected to m
def: connectedElements(m:MM::Method,
     e:Set(TupleType(m1:MM::Method, m2:MM::Method)))
      :Set(MM::Method)
= e->iterate(t:TupleType(m1:MM::Method, m2:MM::Method);
  result3:Set(MM::Method) = Set |
  if m = t.m1 then
     result3->including(t.m2)
  else
    if t.m2  = m then
       result3->including(t.m1)
    else result3
```

```
    endif
  endif )

-- Returns the LCOM1 value for the Class c
context Cohesion::LCOM1(c:MM::Class):Real
body: let methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.implemented_method) in
      getP1(c, methodPairs)->size()/2

-- Returns the LCOM2 value for the Class c
context Cohesion::LCOM2(c:MM::Class):Real
body: let methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.implemented_method),
          p:Real = getP2(c, methodPairs)->size()/2,
          q:Real = getQ(c, methodPairs)->size()/2 in
      if p > q then p - q
      else 0
      endif

-- This is a modified LCOM2 definition
context Cohesion::NewLCOM2(c:MM::Class):Real
body: let methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.implemented_method),
          p:Real = getP1(c, methodPairs)->size(),
          q:Real = getQ(c, methodPairs)->size() in
      if p > q then p - q
      else 0
      endif

-- Returns the LCOM3 value for the Class c
context Cohesion ::LCOM3(c:MM::Class):Real
body: let v:Set(MM::Method) = c.M_i(),
          methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.implemented_method),
          e:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = getQ(c, methodPairs) in
      self.constructConnectedComponents(v, e)->size()

-- Returns the LCOM4 value for the Class c
context Cohesion::LCOM4(c:MM::Class):Real
body: let v:Set(MM::Method) = c.M_i(),
          methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.implemented_method),
          e:Set(TupleType(m1:MM::Method,
          m2:MM::Method)) = getE(c, methodPairs) in
      self.constructConnectedComponents(v, e)->size()

-- Returns the C value for the Class c-for Classes with LCOM4 = 1
context Cohesion::Co(c:MM::Class):Real
body: let methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.M_i()),
          e:Real = self.getE(c, methodPairs)->size()/2,
          v:Real = c.M_i()->size() in
      (2* ( (e - (v-1)) / ((v-1)*(v-2)) ) )

-- This is a redefinition of Co
context Cohesion::NewCo(c:MM::Class):Real
```

```
body: let methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(c.M_i()),
          e:Real = self.getE(c, methodPairs)->size()/2,
          v:Real = c.M_i()->size() in
      (e)/(v*(v-1))

-- Returns the LCOM5 value for the Class c
context Cohesion::LCOM5(c:MM::Class):Real
body: let a:Real = c.A_i()->size(), m:Real = c.M_i()->size() in
      if a = 0.0 then
         0.0
      else
        ((m-((1.0/a)*(self.sumInts(c.A_i()
        ->collect(i:MM::Attribute|i.referenced_by
        ->intersection(c.M_i())->size())) )) )
        /(m-1.0))
      endif

-- Redefined LCOM5 so not an inverse cohesion measure)
context Cohesion::NewCoh(c:MM::Class):Real
body: let a:Real = c.A_i()->size(), m:Real = c.M_i()->size() in
      if m = 0.0 or a = 0.0 then
         0.0
      else
        self.sumInts(c.A_i()->collect(i:MM::Attribute|i.referenced_by
                          ->intersection(c.M_i())->size()))/(m*a)
      endif

-- Returns the TCC value for the Class c
context Cohesion::TCC(c:MM::Class):Real
body: let i:Set(MM::Method) = c.M_i()->intersection(c.M_pub()),
          methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(i), m:Integer = i->size() in
      (2*((methodPairs->select(t|t.m1 <> t.m2 and
      self.cau(t.m1, t.m2, c))->size()/2) / (m*(m-1)) ) )

-- Returns the LCC value for the Class c
context Cohesion::LCC(c:MM::Class):Real
body: let i:Set(MM::Method) = c.M_i()->intersection(c.M_pub()),
          methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method))
          = self.constructMethodPairs(i), m:Integer = i->size() in
      (2*((methodPairs->select(t|t.m1 <> t.m2
      and self.cau_(t.m1, t.m2, c))->size()/2) / (m*(m-1)) ) )

-- Returns the ICH value for the Class c
context Cohesion::ICH(c:MM::Class):Real
body: self.sumReals(c.M_i()->collect(m:MM::Method|self.ICH(m,c)))

-- Returns the ICH value for the Method m
context Cohesion::ICH(m:MM::Method, c:MM::Class):Real
body: sumInts((c.M_new()->union(c.M_ovr()))
                       ->collect(i:MM::Method |
                         (1+i.Par()->size())*(m.NPI(i))))
```

D Auxillary Definitions

```
-- Returns the maximum element in a set of reals
def: max(s:Set(Real)):Real
= s->iterate(elem:Real; result:Real = -1|result.max(elem))

-- Sums a list of reals
def: sumReals(set:Bag(Real)):Real
= set->iterate(i:Real; sum: Real = 0.0|sum + i)

-- Sums a list of ints
def: sumInts(set:Bag(Integer)):Real
= set->iterate(i:Integer; sum: Integer = 0|sum + i)

-- Returns the set of method pairs for a given set of methods
def: constructMethodPairs(methods:Set(MM::Method))
     :Set(TupleType(m1:MM::Method, m2:MM::Method))
= methods->collect(m1|methods
         ->collect(m2|Tuplem1:MM::Method = m1, m2:MM::Method = m2))
         ->asSet()

-- Returns a set containing the attributes referenced by
-- the implemented methods of the Class c
def: getReferencedAtts(c:MM::Class):Set(MM::Attribute)
= c.M_i()->collect(m:MM::Method|m.AR())->asSet()

-- Returns a set containing (pairs of) methods of the Class c that
-- do not directly access any common attributes of c (for LCOM1)
def: getP1(c:MM::Class,
     methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method)))
     :Set(TupleType(m1:MM::Method, m2:MM::Method))
= methodPairs->select(t| ( t.m1 <> t.m2 )
  and ( (t.m1.AR())->intersection(t.m2.AR())
                   ->intersection(c.A_i())->isEmpty()))

-- Returns a set containing the (pairs of) methods of the
-- Class c that do not directly access any common attributes of c or
-- the empty set if all methods of c do not reference any attributes
def: getP2(c:MM::Class,
     methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method)))
     :Set(MM::Attribute)
= if self.getReferencedAtts(c)->size() = 0 then
     Set{}
  else
    methodPairs->select(t| ( t.m1 <> t.m2 )
    and ((t.m1.AR())->intersection(t.m2.AR())
                    ->intersection(c.A_i())->isEmpty()))
  endif

-- Returns a set containing the (pairs of) methods of the
-- Class c that directly access at least one same attribute of c
def: getQ(c:MM::Class,
     methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method)))
     :Set(TupleType(m1:MM::Method, m2:MM::Method))
= methodPairs->select(t|t.m1 <> t.m2 and
  not (t.m1.AR()->intersection(t.m2.AR())
```

```
                ->intersection(c.A_i())->isEmpty()) )

-- Returns a set containing the (pairs of) methods of the
-- Class c that directly access at least one same attribute of c
def: getE(c:MM::Class,
     methodPairs:Set(TupleType(m1:MM::Method, m2:MM::Method)))
     :Set(TupleType(m1:MM::Method, m2:MM::Method))
= methodPairs->select(t|t.m1 <> t.m2
  and (t.m1.AR()->intersection(t.m2.AR())
                ->intersection(c.A_i())->notEmpty()
       or m1.SIM()->includes(m2) or m2.SIM()->includes(m1) ))
```

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Int. Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 436–450, 2007.

[2] A. L. Baroni, S. Braz, and F. B. e Abreu. Using OCL to formalize object-oriented design metrics definitions. In *ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.

[3] L. Briand, J. Daly, and J. Wuest. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[4] L. Briand, J. Daly, and J. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[5] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Intl. Symposium on Software Reliability Engineering*, pages 85–94, Raleigh, NC, Nov. 2006.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] K. Ehrig, J. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from metamodels. In *Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 156–170, 2006.

[8] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy. A novel approach to formalize object-oriented design metrics. In *Evaluation and Assessment in Software Engineering*, Keele, UK, Apr. 2005.

[9] N. Fenton and S. Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Intl. Thompson Computer Press, 1996.

[10] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[12] D. Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, 2006.

[13] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *3rd International Workshop on Critical Systems Development with UML*, Lisbon, Portugal, Oct. 2004.

[14] J. A. McQuillan and J. F. Power. Experiences of using the Dagstuhl Middle Metamodel for defining software metrics. In *Intl. Conference on Principles and Practices of Programming in Java*, pages 194–198, Germany, 2006.

[15] J. A. McQuillan and J. F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming*, Nantes, France, July 4 2006.

[16] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[17] A. Naumenko and A. Wegmann. A metamodel for the Unified Modeling Language. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, volume 2460, pages 2–17. Springer, 2002.

[18] Object Management Group. Meta Object Facility (MOF) Core Specification v2.0. Doc # formal/06-01-01, Jan. 2006.

[19] Object Management Group. UML Superstructure Specification v2.1.1. Doc # formal/07-02-05, Feb. 2007.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting your models ready for MDA*. Addison-Wesley, 2003.

[21] F. G. Wilkie and T. J. Harmer. Tool support for measuring complexity in heterogeneous object-oriented software. In *IEEE Intl. Conference on Software Maintenance*, pages 152–161, Montréal, Canada, Oct. 2002.

[22] A. Zito and J. Dingel. Modeling UML2 package merge with Alloy. In *First Alloy Workshop of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, Portland, OR, Nov. 2006.