Foundations for Defining Software Metrics

Rüdiger Lincke and Welf Löwe

School of Mathematics and Systems Engineering,
Växjö University, 351 95 Växjö, Sweden
{rudiger.lincke|welf.lowe}@msi.vxu.se

Abstract. Ambiguous metric definitions lead to incomparable implementation variants in tools. In this paper, we propose an unambiguous description framework for software metrics, which includes an extensible meta-model and language mapping definitions. We start from the Dagstuhl Middle Metamodel (DMM), an existing and well-defined meta-model for reverse engineering. As examples, we define some software quality metrics based on this meta-model, as well as language mappings for Java and UML/XMI. Meta-models, language mappings, and metrics have been implemented using the definitions with our description framework as the specification basis. We suggest that our description framework eliminates existing ambiguities in software metric definitions, simplifying the implementation of metrics in their original sense, an allowing for the definition of standardized metrics.

1 Introduction

The development and maintenance of software is expensive. Cost estimations of resources and time range from 50% to 80% of the total costs of ownership of a software system [1]. Quality Assurance and Quality Management processes and practices, e.g., ISO9000 family [2] and CMMI [3], are used in industry in order to actively improve the quality of processes and resulting products. Most of these processes depend on the collection of qualitative and quantitative data used for identifying room for improvement and for controlling the success of applied measures. Metrics for measuring software attributes and quality models organizing them in a meaningful way have been researched. These efforts resulted in various metrics suites, among others Chidamber and Kemerer's Metrics Suite for Object Oriented Design [4] and "The MOOD Metrics Set" [5] and in industrial standards like ISO 9126 [6] describing a Software Quality Model.

Furthermore, a large set of tools has been developed for facilitating the evaluation of these metrics on several different programming languages, including C/C++ and Java, and other high-level representations, like UML. Yet the results provided by the different tools are difficult to compare, since there is, so far, no *unambiguous* description framework for metrics. Unambiguous means that given a metric definition and a source program or specification the result of that measurement is uniquely defined. This hampers the implementation of

metrics "by the book" and the discussion of variants, let away scientific comparative and repetitive studies. Hence, common validation efforts, e.g., the FAMOOS project [7], are of limited significance.

An early reason for ambiguities was that data models (meta-models) – a basis for the metrics definitions – were implicit and not included in the definitions [8]. This problem has been noted among others by Churcher and Shepperd, who identified ambiguities in the suite of metrics proposed by Chidamber and Kemerer [4]. They even needed a dialog with the authors of the metrics to interpret them in a "correct" way [9]. The scientific community has addressed this source of ambiguity by proposing common and formally defined meta-models. Metrics were precisely described on these models according to the individual interpretations. Examples are the FAMIX meta-model [10], the Data Model for Object-Oriented Design Metrics [11], and the Dagstuhl Middle Metamodel [12]. Refer to the related work section for more details. Yet we see two limitations:

1. The meta-models themselves are ambiguous, because it is not specified how the information from a software system is mapped to the meta-model. Because of this missing language binding, two metric tools – for the same source language, using the same meta-model and metric definition – may measure different things in the same software system.[1]
2. Even though meta-models are extensible, so far none of them allows to control effects of extension on metrics definitions. It should be controllable if they deliver the same results, after a meta-model extension or, otherwise, if a re-definition becomes necessary.

Our proposal for overcoming ambiguity is a description framework including meta-model, language binding, and metric definitions. It uses the Dagstuhl Middle Metamodel[2] [12] as its initial meta-model. We apply the extension method for meta-models proposed by Strein et al. in [13] to add controlled extensibility.

Section 2 summarizes our description framework [13]. Section 3 presents a full definition of DMM using this framework, including language mappings for Java and UML. Section 4 completes description framework for defining metrics and some examples of metric definitions. Section 5 discusses related work. Finally, Section 6 concludes the results.

2 The Description Framework

Our description framework is based on the extension method for meta-models in [13], which assumes an architecture for constructing, capturing and accessing models for software systems. Whereas [13] describes the Meta-Meta-Model and Meta-Model level, we describe in this paper an instantiation (Model layer) consisting of the following components (cf. Fig. 1): (1) Different front-ends extracting

[1] For the language-transparent middleware Corba, the language binding is part of the standard to avoid this kind incompatibilities.

[2] We preferred to select the DMM over defining our own meta-model, since this model is rather complete, and already accepted.

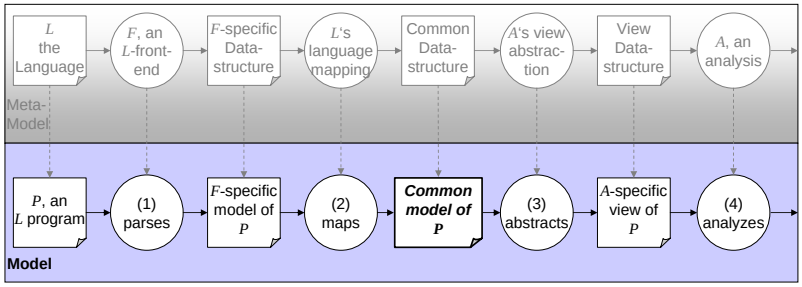

Fig. 1. Process of information extraction, mapping to language-independent representation, abstraction and analysis on mode. Taken from [13].

information form source codes or other program representations. (2) Data stores, capturing program information relevant for later analysis, e.g., in metrics. This information is mapped from language-specific to language-independent formats (we use DMM for the latter). (3) Abstractions computing *views* on the model that are specific for a set of analyses, allowing to control effects of model extension. (4) Different *analysis* accessing the model via their respective abstract views. The *Common model of P* is the common denominator in the related work. We need the other components to be unambiguous.

The representations of relevant information in source codes, data store, and views, respectively, are defined in source-language-specific, common, and metric-specific meta-models. Each meta-model M is defined as pair $M = (G, R)$, a tree grammar G and a set of semantic relations R. We prefer this notation over other notations, since it is closer to the notation used for specifying programming languages, making the description of the mapping straight forward. Nevertheless any other notation like MOF notations including UML could be used as well. A tree grammar $G = (T, P, prog)$ consists of the set of model entities T (node types), a set of productions defining the possible tree structures P (cf. Table 1), and the root node type of the structural containment trees $prog \in T$. Productions $p \in P$ have the form: $t ::= expr$, where $t \in T$ and $expr$ is an expression over $t \subseteq T$. Expressions are either sequences $(t_1 \dots t_k)$, iterations (t^*), or alternatives $(t_1| \dots |t_k)$ with the obvious semantics [13].

A type $t \in T$ that is occurring on the left-hand side of an alternative production $t_1| \dots |t_k$ is called *super type* of types $t_1, \dots, t_k$. This defines a *specialization hierarchy* $\prec$ defined over T. We interpret $\prec$ on node type in the following way: let $t, t' \in T$ and $t \prec t'$, then $n \in t \Rightarrow n \in t'$.

R (cf. Table 2) denotes a set of semantic relations over model entities with $R = \{R_1, \dots, R_n\}$, having for each $R_i, 1 \leq i \leq n$ subsets of entities T defined, i.e., $R = T_1 \times T_2$. Additionally, there is a specialization hierarchy $\prec$ (cf. Table 3) over the semantic relations.

For mapping the model entities and semantic relations from the model specific source language `lang` to the common meta-model, i.e., the language binding, source-language-specific mapping functions α^{lang} are defined (cf. Table 6).

Model	::=	ASourceObject* AModelObject*
ASourceObject	≻	ASourcePart | ASourceUnit
ASourcePart	≻	SourcePart | MacroExpansion | MacroArgument | ADefinition | AResolvable | Comment
ADefinition	≻	MacroDefinition | Definition
AResolvable	≻	Declaration | Reference | Resolvable
ASourceUnit	≻	SourceFile | SourceUnit
AModelObject	≻	ModelObject | AModelElement
AModelElement	≻	ModelElement | Package | AStructuralElement | ABehaviouralElement
AStructuralElement	≻	AType | AValue
AType	≻	Type | AStructuredType | CollectionType | EnumeratedType
AStructuredType	≻	StructureType | Class
Class	::=	Field* Method*
AValue	≻	Value | EnumerationLiteral | AVariable
AVariable	≻	Variable | Field | FormalParameter
EnumerationType	::=	EnumerationLiteral*
ABehaviouralElement	≻	ExecutableValue | Method | Routine
Method	::=	FormalParameter*
ExecutableValue	::=	FormalParameter*
Routine	::=	FormalParameter*

Table 1. Dagstuhl Middle Metamodel, productions P and specialization hierarchy

Similar mapping functions α^{metric} are used when defining an abstraction from common model to a view specific *metric*.

Metric-specific views allow a controlled extensibility of the meta-model. If a view is guaranteed not to be effected by a meta-model extension, the metrics does not need to be changed. The following conditions make extensions safe for a metric analysis A: (a) Adding a *new* type to a sequence expression on the right hand side of a production. (b) Adding an *existing* node type t to a sequence provided that no node type relevant for A can transitively be derived from t. (c) Introducing a new production $t ::= ...$ provided that no node type relevant for A can transitively be derived from t. (d) Adding a new relation provided none of its transitive super types is relevant for A.

In all these cases, the nodes newly introduced to the data model will be filtered in the view mapping and the relations will be attached to the original node types. Consequently, if a meta-model change is not *safe* for an analysis A, we need to check and potentially adapt A.

For details of the meta-model specifications, their mapping specifications and the corresponding mapping of models, and for their safe extension, we refer to [13]. It remains to discuss how metrics are described based on their metric specific meta-models. We will postpone this until Section 4.

3 DMM using the Description Framework

We use our notation of tree-grammars and relations, to describe the Dagstuhl Middle Metamodel version 0.007. Further, we sketch the meta-models for two front-ends and their mapping to the DMM. Additionally, we describe views on the meta-model that we will need for three (example) metrics definitions.

3.1 The Dagstuhl Middle Metamodel 0.007

Originally, the Dagstuhl Middle Metamodel 0.007 (DMM) was defined in an UML notation, cf. [12]. We use our model description $M = (G, R)$ instead, since

Accesses	: ABehaviouralElement × AStructuralElement
Contains	: SourceObject × SourcePart, Package × ModelElement
Declares	: SourceObject × ModelObject
Defines	: SourceObject × ModelObject
Describes	: SourceObject × Comment
HasValue	: Variable × Value
Imports	: Class × Package
Includes	: SourceFile × SourceFile
InheritsFrom	: Class × Class
Invokes	: ABehaviouralElement × ABehaviouralElement
IsActualParameterOf	: ModelElement × Invokes
IsDefinedInTermsOf	: Type × Type
IsEnumerationLiteralOf	: EnumerationLiteral × EnumeratedType
IsExpansionOf	: MacroDefinition × MacroExpansion
IsFieldOf	: Field × StructuredType
IsMethodOf	: Method × Class
IsOfType	: Value × Type
IsParameterOf	: FormalParameter × BehaviouralElement
IsReturnTypeOf	: Type × BehavioralElement
IsSubpackageOf	: Package × Package

Table 2. Dagstuhl Middle Metamodel, binary semantic relations in R

Relationship	≻ ModelRelationship, SourceRelationship, SourceModelRelationship
ModelRelationship	≻ InheritsFrom, IsPartOf, Invokes, IsOfType, Accessess, IsDefinedInTermsOf, IsPartOfSignatureOf, IsActualParameterOf, HasValue, IsSubpackageOf
IsPartOf	≻ IsEnumerationLiteralOf, IsMethodOf, IsFieldOf
InheritsFrom	≻ Extends, Implements
IsPartOfSignatureOf	≻ IsParameterOf, IsReturnTypeOf

Table 3. Dagstuhl Middle Metamodel, specialization hierarchy $\prec$ of semantic relations

this is closer to the notation used for programming languages specifications. This simplifies the definition of the language binding later on.

The root node type of G is *Model*. The productions P of G describe a structural containment relation, denoted by ::= in the productions, and a specialization hierarchy on the meta-model entities T, denoted by $\succ$, cf. Table 1. The structural containment relation is added to the original DMM. In contrast to DMM, we separate abstract and concrete meta-model elements: abstract model element, denoted with "A<*entity name*>", are never instantiated in a concrete model.

Attributes of DMM entireties (unary relations) and binary relations are defined in R as unary and binary relations respectively. Unary relations include: isSubclassable(Class), size(CollectionType), position(FormalParameter), name(MacroArgument | MacroDefinition | ModelObject | AResolvable | ASourceUnit), isConstructor(Method), isDestructor(Method), isAbstract(Method), isDynamicallyBound(Method), isOverrideable(Method), visibility(Field | Method | ModelElement), path(SourceFile), startLine(SourcePart), start(SourcePart), endLine(SourcePart), endChar(SourcePart). Binary relations and their specialization *prec* are listed in Table 2 and 3, respectively.

3.2 Java Specific Model (Recoder Front-end)

This section describes a language-specific representation of the programs that the metrics are applied to. Therefore, we use a meta-model of the compiler front-end Recoder [14] denoted by $M^{\mathtt{R}} = (G^{\mathtt{R}}, R^{\mathtt{R}})$.[3]

[3] Actually, one should base the language mapping on the grammar and the semantic relations of the Java language specification [15]. For brevity of presentation, we

program^R	::= compilation_unit^R *
compilation_unit^R	::= type^R +
type^R	$\succ$ class^R | interface^R
interface^R	::= method^R *
class^R	::= constructor^R * method^R * field^R * initialization_block^R?
method^R	::= statement^R *
statement^R	$\succ$ assign^R | call_expr^R | create_expr^R | do^R | for^R | if^R | switch^R | while^R
initialization_block^R	::= statement^R *
constructor^R	::= statement^R *
do^R	::= statement^R *
for^R	::= statement^R *
if^R	::= statement^R *
switch^R	::= statement^R *
while^R	::= statement^R *
assign^R	::= expression^R expression^R -- *left-hand side '=' right-hand side*
expression^R	$\succ$ call_expr^R | create_expr^R | read_expr^R | write_expr^R | ...
call_expr^R	::= expression^R * -- *first designator then actual parameters*
create_expr^R	::= expression^R * -- *actual parameters*

Table 4. Recoder front-end, productions P^R

call^R	: call_expr^R $\times$ method^R
create^R	: create_expr^R $\times$ method^R
extends^R	: interface^R $\times$ interface^R
extends^R	: class^R $\times$ class^R
implements^R	: class^R $\times$ interface^R
field_access^R	: read_expr^R $\times$ field^R
field_access^R	: write_expr^R $\times$ field^R
type_ref^R	: expression^R $\times$ type^R

Table 5. Recoder front-end, semantic relations R^R

α^R(program^R)	$\mapsto$ Model
α^R(compilation_unit^R)	$\mapsto$ SourceUnit
α^R(class^R)	$\mapsto$ Class
α^R(interface^R)	$\mapsto$ Class
α^R(constructor^R)	$\mapsto$ Method, Method.isConstructor = true
α^R(method^R)	$\mapsto$ Method, Method.isConstructor = false
α^R(field ^{R})	$\mapsto$ Field
α^R(initialization_block ^{R})	$\mapsto$ Method, Method.isConstructor = true
α^R(call^R)	$\mapsto$ Invokes
α^R(create^R)	$\mapsto$ Invokes
α^R(extends^R)	$\mapsto$ InheritsFrom, InhertisFrom.inheritanceType = Extends
α^R(implements^R)	$\mapsto$ InheritsFrom, InhertisFrom.inheritanceType = Implements
α^R(field_access^R)	$\mapsto$ Accesses
α^R(type_ref^R)	$\mapsto$ IsOfType

Table 6. Mapping functions α^R

The model entities T^R of the grammar G^R are implicitly defined by the productions P^R, which in turn define the containment structure of the model entities and their specialization hierarchy, cf. Table 4. The root node type of G^R is *program*. The list of unary semantic relations (intrinsic node attributes like names, positions, and visibility) is straight forward and therefore omitted. Binary semantic relations R^R are shown in Table 5. There is no particular type hierarchy $\prec^R$ for relations defined.

The mapping functions for mapping the language-specific types T^R and relations R^R to the common meta-model are given in Table 6. The mapping of actual models filters nodes of types t^R with $\alpha(t^R)$ not defined. Otherwise it keeps the

choose to start from a front-end implementing Java instead. This does not compromise our goal of unambiguous language bindings since compiler verification is well understood for front-ends [16].

model^{U}	$::= \text{cluster}^{\text{U}} *$	$\text{dependency}^{\text{U}}$	$\succ^{\text{U}} \text{association}^{\text{U}}$
$\text{cluster}^{\text{U}}$	$::= \text{module}^{\text{U}} * \text{class}^{\text{U}} *$		$\succ^{\text{U}} \text{generalization}^{\text{U}}$
module^{U}	$::= \text{class}^{\text{U}} *$		$\succ^{\text{U}} \text{message}^{\text{U}}$
class^{U}	$::= \text{attribute}^{\text{U}} * \text{operation}^{\text{U}} *$		$\succ^{\text{U}} \text{binding}^{\text{U}}$
$\text{operation}^{\text{U}}$	$::= \text{parameter}^{\text{U}} *$		$\succ^{\text{U}} \text{uninterpretedaction}^{\text{U}}$
		$\text{association}^{\text{U}}$	$\succ^{\text{U}} \text{type\_ref}^{\text{U}}$

Table 7. UML-specific productions P^{U} (left) and specialization hierarchy $\prec^{\text{U}}$ on relations (right).

$\text{generalization}^{\text{U}}$	$: \text{class}^{\text{U}} \times \text{class}^{\text{U}}$
$\text{dependency}^{\text{U}}$	$: \text{class}^{\text{U}} \times \text{class}^{\text{U}}$
	$: \text{class}^{\text{U}} \times \text{operation}^{\text{U}}$
	$: \text{attribute}^{\text{U}} \times \text{class}^{\text{U}}$
	$: \text{module}^{\text{U}} \times \text{module}^{\text{U}}$
	$: \text{cluster}^{\text{U}} \times \text{module}^{\text{U}}$
	$: \text{module}^{\text{U}} \times \text{cluster}^{\text{U}}$
	$: \text{operation}^{\text{U}} \times \text{class}^{\text{U}}$
$\text{association}^{\text{U}}$	$: \text{class}^{\text{U}} \times \text{class}^{\text{U}}$
$\text{message}^{\text{U}}$	$: \text{class}^{\text{U}} \times \text{operation}^{\text{U}}$
$\text{type\_ref}^{\text{U}}$	$: \text{attribute}^{\text{U}} \times \text{class}^{\text{U}}$
	$: \text{parameter}^{\text{U}} \times \text{class}^{\text{U}}$
	$: \text{operation}^{\text{U}} \times \text{class}^{\text{U}}$
uninterpretedaction	$: \text{class} \times \text{class}$
$\text{binding}^{\text{U}}$	$: \text{class}^{\text{U}} \times \text{class}^{\text{U}}$
	$: \text{module}^{\text{U}} \times \text{module}^{\text{U}}$
	$: \text{cluster}^{\text{U}} \times \text{cluster}^{\text{U}}$

$\alpha^{\text{U}}(\text{attribute}^{\text{U}})$	$\mapsto$ Field
$\alpha^{\text{U}}(\text{class}^{\text{U}})$	$\mapsto$ Class
$\alpha^{\text{U}}(\text{model}^{\text{U}})$	$\mapsto$ Model
$\alpha^{\text{U}}(\text{operation}^{\text{U}})$	$\mapsto$ Method
$\alpha^{\text{U}}(\text{cluster})$	$\mapsto$ Package
$\alpha^{\text{U}}(\text{module})$	$\mapsto$ Package
$\alpha^{\text{U}}(\text{parameter}^{\text{U}})$	$\mapsto$ FormalParameter
$\alpha^{\text{U}}(\text{association}^{\text{U}})$	$\mapsto$ FieldAccess
$\alpha^{\text{U}}(\text{dependency}^{\text{U}})$	$\mapsto$ ModelRelationship
$\alpha^{\text{U}}(\text{generalization}^{\text{U}})$	$\mapsto$ Inheritance
$\alpha^{\text{U}}(\text{message}^{\text{U}})$	$\mapsto$ Invokes

Table 8. Semantic relations R^{U} (left) and mapping α^{U} to the common meta-model (right).

containment structure of the language-specific model also in the common model, but some transitive children in the language-specific model become direct children in the common model. Tuples of the language-specific $rel^{\text{R}} : t_1^{\text{R}} \times t_2^{\text{R}}$ are mapped to the corresponding tuple of $\alpha^{\text{R}}(rel^{\text{R}}) : t_1 \times t_2$. If nodes of a source tuple are filtered, the target tuple nodes become the next suiting parent in the target containment structure. For example, $\alpha^{\text{R}}(\text{ call }^{\text{R}} : \text{call\_expr}^{\text{R}} \times \text{method}^{\text{R}}) \mapsto$ Invokes : ABehaviouralElement $\times$ ABehaviouralElement, and call^{R} nodes are filtered. Hence, a source tuple $\text{call}^{\text{R}}(aCallExpr, aMethod)$ with $aCallExpr$ contained in a method $anotherMethod$ of the language-specific model is mapped to a tuple $\text{Invokes}(anotherMethod', aMethod')$ with $anotherMethod'$ and $aMethod'$ the target of $anotherMethod$ and $aMethod$, respectively, in the common model.

Currently, our Java meta-model covers not all parts of the Java Language Specification. We include in our mapping only model information required for the metric-relevant parts of the static structure of a software system. This is a *lazy approach*; we can easily extend the model.

3.3 UML-Specific Model

This section describes a language-specific (simple) UML class diagram meta-model $M^{\text{U}} = (G^{\text{U}}, R^{\text{U}})$ and its language binding. The model entities T^{U} and the productions P^{U} describing the containment structure over the model entities are defined in Table 7 (left). The root node type is model^{U}. There is no particular specialization hierarchy $\prec^{\text{U}}$ defined over these types. The (binary) semantic relations R^{U} are shown in Table 8 (Left). The specialization hierarchy is defined in Table 7 (right).

The mapping functions for mapping the front-end model specific types $T^{\mathtt{U}}$ and relations $R^{\mathtt{U}}$ into the common meta-model are given in Table 8 (right).

As with the Java Model we follow the *lazy approach*, implementing elements as needed. Therefore our UML meta-model covers not all parts of the OMG meta-model definition for UML. We focus only on model information related to Class and Sequence diagrams, relevant for representing the static structure of the software system included in our mapping. Layout related diagram information is ignored. Other diagram types, like Use-Case Diagrams, are currently not relevant and therefore not included.

4 Definition of Metrics

This section completes our description framework introduced in Section 2 by defining a schema for describing the actual metrics, cf. Section 4.1. Then we exemplify the schema by applying it to three standard metrics, cf. Section 4.2, which also includes an example for safe extension of the common meta-model. Currently we are focusing on Object-Oriented and Non-Object-Oriented Code Metrics. We might need to extend this Definition Framework in the future, to include requirements, use-cases, test coverage, etc. metrics.

4.1 Metrics Definition Framework

For the metrics themselves, we use the following description schema, which completes our description framework:

Description A textual description of the metric and what it measures.
Scope Identifies the program element the metric applies to. This could be *Model* for the whole software system, *Package* for entities of a package or a directory, *Class, SourceFile, SourceUnit, ModelObject* for classes, modules and files, respectively, and *Routine, Method* for routines and methods, respectively.
View Definition of the *view*, describing its relevant elements and semantic relations and the tree grammar.
Definition A formal definition of the metric based on the view.
Scale Defines the scale type of the metric. It is one of *Absolute*, *Rational*, *Interval*, *Ordinal*, *Nominal*.
Domain Defines the domain of metric values, i.e., the co-domain of the metric analysis.

The definition of the metrics is based on the common meta-model as defined before. For each metric, we define a *view* further abstracting from the common meta-model and providing exactly the information required by that metric. The view is used for the actual metric definition. This approach makes the metric definitions independent from changes of the common meta-model. Formally, views of a metric analysis A are again defined as pairs $V^A = \left(G^A, R^A\right)$ and bound to the common model with a mapping specification α^A. Again, G^A is a tree grammar specifying the set of view entities and their structural containment requited by A. R^A is a set of semantic relations over view entities required by A.

The construction of concrete view models follows the same principles as the abstractions from front-end specific to common models: we ignore some common meta-model entity types, which leads to a filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types and attach remaining relations defined over filtered nodes to the relevant ancestors of those nodes, as described in detail in [13].

To simplify metric definitions, we define some utility operation(s): $succ(e, r)$ [$succ^*(e, r)$, $succ^+(e, r)$] denotes the set of direct [transitive, transitive excluding e] successors of a node e over edges (relations) of type r. $|S|$ denotes the number of elements in a set S.

4.2 Example Metric Definitions

We define three metrics using the description framework above: Number Of Children [4], Coupling Factor [5], and McCabe's Cyclomatic Complexity [17]. These metrics are unambiguously defined, because documentation exists for: (1) a meta-model used for describing metrics, (2) a mapping from the front-end to the meta-model, (3) views onto the meta-model filtering unnecessary information, and (4) unambiguous description of the elements analyzed using relational algebra. Therefore, it is perfectly clear how metrics values are calculated, preparing ground for comparable studies and the validation and standardization of metrics.

Number Of Children (*NOC*) works with all instances of a common meta-model, regardless if they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the NOC specific view).

Description NOC is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. NOC measures how many classes inherit directly methods or fields from a super-class. NOC is only applicable to object-oriented systems.

Scope Class

View $V^{NOC} = (G^{NOC}, R^{NOC})$

- Grammar $G^{NOC} = (\{\text{class}^{NOC}\}, \emptyset, \text{class}^{NOC})$
- Relations $R^{NOC} : \{\text{inheritance}^{NOC} : \text{class}^{NOC} \times \text{class}^{NOC}\}$
- Mapping α^{NOC}:

$$\alpha^{NOC}(\text{Class}) \mapsto \text{class}^{NOC}$$
$$\alpha^{NOC}(\text{Inheritance}) \mapsto \text{inheritance}^{NOC}$$

Definition The NOC value of a class $c \in class^{NOC}$ is defined as:

$$NOC(c) := \left| succ(c, \text{inheritance}^{NOC}) \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Coupling Factor (*CF*) also works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Coupling Factor (CF) measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes. CF is primarily applicable to object-oriented systems.

Scope Package

View $V^{CF} = (G^{CF}, R^{CF})$

- Grammar $G^{CF} = (\{\text{package}^{CF}, \text{class}^{CF}\}, P^{CF}, \text{package}^{CF})$
- Productions $P^{CF} = \{\text{package}^{CF} ::= \text{class}^{CF\ *}\}$
- Relations $R^{CF} : \{\text{coupling}^{CF}\}$[4]
- Mapping α^{CF}:

$$\begin{aligned}
\alpha^{CF}(\text{Class}) &\mapsto \text{class}^{CF} \\
\alpha^{CF}(\text{Package}) &\mapsto \text{package}^{CF} \\
\alpha^{CF}(\text{Invokes}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{Accesses}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{IsOfType}) &\mapsto \text{coupling}^{CF}
\end{aligned}$$

Definition The CF value of a package $p \in \text{package}^{CF}$ is defined:

$$\begin{aligned}
Classes(p) &= succ^*(p, \text{contains}^{CF}) \cap \text{class}^{\texttt{CF}} \\
&\quad \textit{-- set of classes contained in } p \\
Coupled(p, c) &= succ(c, \text{coupling}^{CF}) \cap Classes(p) \\
&\quad \textit{-- set of classes contained in } p\textit{, which } c \textit{ is coupled to} \\
CF(p) &= \frac{\sum_{c \in Classes(p)} \mid Coupled(p, c) \mid}{\mid Classes(p) \mid^2 - \mid Classes(p) \mid}
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

[4] The structural contains^{CF} relation is implicitly defined by the productions P^{CF}.

Method	::= FormalParamter* Statement*	$\alpha^{\mathtt{R}}(\mathrm{do}^{\mathtt{R}})$	$\mapsto$ Loop
Statement	::= Do | For | If | Switch | While	$\alpha^{\mathtt{R}}(\mathrm{for}^{\mathtt{R}})$	$\mapsto$ Loop
Do	::= Statement*	$\alpha^{\mathtt{R}}(\mathrm{while}^{\mathtt{R}})$	$\mapsto$ Loop
For	::= Statement*	$\alpha^{\mathtt{R}}(\mathrm{if}^{\mathtt{R}})$	$\mapsto$ Switch
If	::= Statement*	$\alpha^{\mathtt{R}}(\mathrm{switch}^{\mathtt{R}})$	$\mapsto$ Switch
Switch	::= Statement*		
While	::= Statement*		

Table 9. Extended Common Meta-Model (left) and new mapping functions $\alpha^{\mathtt{R}}$ for Java (right).

McCabe Cyclomatic Complexity (*CC*) cannot be calculated on the original meta-model, since CC requires control flow statements. Therefore, the common-model has to be extended first, cf. Table 9 (left). In particular, we did the following modifications of the model $M = (G, R)$ defined in Section 3.1. According to extension Rule 1 (cf. Section 2) we add a new entity type Statement to the production: Method ::= FormalParamter *. According to extension Rule 3, we add a Statement including new entity types for loops and branches Loop and Switch. No new semantic relations were added.

The mapping functions α^R for the Java front-end was extended as well, cf. Table 9 (right), to map the corresponding Java entities, which already existed in the language-specific meta-model.

After extension, CC can be computed on instances of a common meta-model, as long as the required types are provided by the front-end. A UML front-end, e.g., would not construct nodes of required type. Still, this front-end works with the new meta-model and the metric is unambiguously defined.

Description CC is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.
Scope Method
View $V^{CC} = (G^{CC}, \emptyset)$.

- Grammar $G^{CC} = (T^{CC}, P^{CC}, \mathrm{method}^{CC})$
- Entities $T^{CC} = \{\mathrm{method}^{CC}, \mathrm{ctrl\_stmt}^{CC}\}$
- Productions P^{CC}:

$$\begin{aligned} \mathrm{method}^{CC} &= \mathrm{ctrl\_stmt}^{CC}\ * \\ \mathrm{ctrl\_stmt}^{CC} &= \mathrm{ctrl\_stmt}^{CC}\ * \end{aligned}$$

- Mapping α^{CC}:

$$\begin{aligned} \alpha^{CC}(\mathrm{Method}) &\mapsto \mathit{method}^{CC} \\ \alpha^{CC}(\mathrm{Switch}) &\mapsto \mathrm{ctrl\_stmt}^{CC} \\ \alpha^{CC}(\mathrm{Loop}) &\mapsto \mathrm{ctrl\_stmt}^{CC} \end{aligned}$$

Definition The CC value of a method $m \in \mathit{method}^{CC}$ is defined as:

$$CC(m) := \left| \mathit{succ}^{+}\left(m, \mathrm{contains}^{CC}\right) \right|$$

Scale Absolute.
Domain Integers in $1..\infty$.

5 Related work

The need for model independent metrics has been addressed by the scientific community in several efforts, defining meta-models for the purpose of metrics evaluation and reverse engineering.

The FAMIX meta-model developed by Lanza and Ducasse in the context of the European Esprit Project FAMOOS [7] resulted in the creation of the Moose Re-engineering Environment, which is based on the language independent FAMIX meta-model [10]. The authors restricted themselves to the field of object-oriented re-engineering and object-oriented metrics, suitable for metrics covering most of the relevant aspects of object-oriented software. In practice this framework deals with information extracted from software written in C++, Java, Smalltalk and Ada [18], mapping the source code to a language independent representation. The framework contains more than 50 different metrics of which are ca. 30 language-independent. The language dependent metrics are explicitly marked as such. The computation of the language independent metrics is based on the repository of software entities and is therefore by definition language independent. As stated in [10] coupling and cohesion metrics are currently not supported by this framework which is a result from the lack of consensus on how to define many of the metrics in this category. Even though the FAMIX meta-model is defined in detail, the mapping of the single language elements in form of front-end (or the context of FAMIX language plug-ins) is not explicitly provided allowing ambiguity in the interpretation of the model elements.

Abounder and Lamb proposed a data model for object-oriented (OO) design metrics [11]. They performed a careful investigation of existing OO metrics and classifications thereof. On this basis, they designed a data model for a database of design information from which most of the proposed OO metrics can be computed. They provided a summary of entities and relationships, but leave the information extraction and the mapping of the elements provided by the different tools/front-ends out of their consideration. Therefore they introduce ambiguity. Extensibility was not explicitly part of their focus.

Briand et al. describe an unified framework for coupling [8] and cohesion [19] measurement in OO systems. Their framework allows for exact definition of metrics using relational algebra, but language mapping is not defined. Extensibility is no focus, new frameworks must be developed for other kinds of metrics.

Wilkie and Harmer created tool support for measuring complexity in object-oriented systems [20]. The meta-model used by the tool is a relational database schema. Analysis (metrics) are described as SQL queries. The extensible meta-model can hold new language elements and new metrics can be added by writing appropriate SQL queries. No language mapping information is provided.

Reißing proposed a formal model for object-oriented design measurement [21]. His model is based on a UML meta-model. Metrics are defined using natural language expressions and the mappings from the front-ends used to the formal model is not part of his work, thus the ambiguity problem is not solved.

El-Wakil et al. present their approach to formalize object-oriented design metrics in [22]. They do not have a explicit meta-model abstracting the in-

formation extracted by front-ends. Instead, they use XQuery as front-end and definition language for the metrics, thus calculating the metrics directly on the front-end model capturing the XMI design documents. Their approach is therefore unambiguous, but limited to a single front-end.

Baroni et al. describe in several papers a UML 1.3 [23] based meta-model and the use of OCL as metrics definition language [24–27]. A tool called FLAME (Formal Library for Aiding Metrics Extraction) implements their efforts and is described in [28]. Goulão et al. [29] builds on their approach with component based metrics and a UML 2.0 [30] based meta-model as foundation for their definitions. Their approach does not explicitly specify a language mapping.

The previous references [8, 19–22, 24–27, 29] are discussed more detailed in [31].

McQuillan et al. recognize the useful mechanisms for the precise definition of software metrics [24–27], and propose a generalization [31]. They extend Baronis' approach by decoupling the metric definitions from the meta-model. They extend the meta-model with a separate metrics package, containing metrics expressed as OCL queries. This approach allows to easy extend their tool, which can in theory by applied to any language, and furthermore, will allow easy re-use of metrics definitions. A prototype tool called DMML (Defining Metrics at the Meta Level) supports their approach. For illustration, formal definitions for the Chidamber and Kemerer metrics suite are presented. This approach is extensible and unambiguous on meta-model level, but still introduces ambiguity by not specifying the mapping from the front-end meta-model to the Dagstuhl Middle Metamodel, they use for capturing model information.

With the Dagstuhl Middle Metamodel (DMM), Lethbridge et al. [12] describe an extensible schema for a static model of software. It captures program level entities and their relationships, rather than full abstract syntax graphs, or architectural abstractions. Their focus is on providing a sound meta-model, not being concerned with meta-meta-models providing methods for controlled extension of the meta-model, or how program information is mapped from front-ends to the DMM. A discussion of further alternative meta-models is given in [13].

6 Conclusion and Future Work

We defined an unambiguous and extensible description framework for software metrics. Its core is the a common meta-model, which is based on the Dagstuhl Middle Metamodel, a well-defined meta-model for re-engineering. It allows for language independent and exact definition of software metrics. We improved on the limitations of existing solutions by describing language mappings for resolving ambiguity. Further we can guarantee the validity of metrics analysis after safe extension of the meta model [13]. We demonstrated feasibility defining two language bindings and three metrics in the framework – many more are implemented in the VizzAnalyzer tool [32]. We suggest that metrics can now be defined in an unambiguous way allowing for experiments and validations. Furthermore, it allows to standardize metrics. As a consequence the prerequisites

for the validation of metrics through empirical studies aiming to show that a metric measures what it is intended to measure are created.

As future work, we are going to use this framework to define and validate the software metrics currently described in literature, cf. [33] for the approach. We hope that we can discuss this definition with the community to be able to agree on a consent in their interpretation. This could be realized in form of a public compendium [34] of metrics, quite in the spirit of "A compendium of NP optimization problems" edited by Pierluigi Crescenzi and Viggo Kann [35].

References

1. Boehm, B.W.: Software Engineering Economics. Prentice Hall (1981)
2. ISO: Iso 9000:2005 "quality management systems - fundamentals and vocabulary" (2005)
3. SEI: Capability maturity model integration (cmmi). http://www.sei.cmu.edu/cmmi/cmmi.html (2006)
4. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. In: IEEE Transactions on Software Engineering. Volume 20 (6). (1994) 476–493
5. e Abreu, F.B.: The mood metrics set. In: Proceedings ECOOP Workshop on Metrics. (1995)
6. ISO: Iso/iec 9126-1 "software engineering - product quality - part 1: Quality model" (2001)
7. Bär, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, S., Lanza, M., Marinescu, R., Nebbe, R., Nierstrasz, O., Przybilski, M., Richner, T., Rieger, M., Riva, C., Sassen, A., Schulz, B., Steyaert, P., Tichelaar, S., Weisbrod, J.: The famoos object-oriented reengineering handbook. http://www.iam.unibe.ch/~famoos/handbook/ (1999)
8. Briand, L., Daly, J., Wüst, J.: A unified framework for coupling measurement in object-oriented systems. In: IEEE Transactions on Software Engineering. Volume 25 (1). (1999) 91–121
9. Churcher, N., Shepperd, M.: Comments on "a metrics suite for object oriented design". In: IEEE Transactions on Software Engineering. Volume 21 (3). (1995) 263–265
10. Lanza, M., Ducasse, S.: Beyond language independent object oriented metrics: Model independent metrics. In: 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering. (2002)
11. Lamb, D., Abounader, J.: Data model for object-oriented design metrics. Technical report, Queen's University, Kingston, ON (1997) ISSN-0836-0227-1997-409.
12. Lethbridge, T.C., Tichelaar, S., Ploedereder, E.: The dagstuhl middle metamodel. In: Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003). Volume 94. (2004) 7–18
13. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An extensible meta-model for program analysis. In: 22nd IEEE International Conference on Software Maintenance. (2006) (accepted for publication).
14. University of Karlsruhe: Recoder. http://recoder.sourceforge.net (2006)
15. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
16. Glesner, S., Goos, G., Zimmermann, W.: Verifix: Konstruktion und architektur verifizierender Übersetzer. IT Information Technology **46**(5) (2004)

17. Watson, A.H., McCabe, T.J.: Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235 (1996)
18. Demeyer, S., Tichelaar, S., Steyaert, P.: Famix - the famoos information exchange model, version 2.0. http://www.iam.unibe.ch/~famoos/FAMIX/ (1999)
19. Briand, L., Daly, J.: A unified framework for cohesion measurement in object-oriented systems. In: Proceedings of the Fourth Conference on METRICS'97. (1997) 43–53
20. Wilkie, F., Harmer, T.: Tool support for measuring complexity in heterogeneous object-oriented software. In: 18th IEEE International Conference on Software Maintenance (ICSM'02). (2002) 152
21. Reißing, R.: Towards a model for object-oriented design measurement. In: Proceedings 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. (2001) 71–84
22. El-Wakil, M., El-Bastawisi, A., Riad, M., Fahmy, A.: A novel approach to formalize object-oriented design metrics. In: Proceedings of Evaluation and Assessment in Software Engineering (EASE), Keele, UK (2005)
23. OMG: Object management group. OMG unified modeling language specification version 1.3 (1999)
24. Baroni, A.: Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel, Faculty of Sciences (Belgium) together with Ecole des Mines de Nantes (France) and Universidade Nova de Lisboa (Portugal) (2002)
25. Baroni, A., Braz, S., e Abreu, F.B.: Using ocl to formalize object-oriented design metrics definitions. In: ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering. Lecture Notes in Computer Science, Springer-Verlag (2002)
26. Baroni, A., e Abreu, F.B.: An ocl-based formalization of the moose metric suite. In: 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. (2003)
27. Baroni, A., Calero, C., Piattini, M., e Abreu, F.B.: A formal definition for object-relational database metrics. In: Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS 2005), Miami, USA (2005)
28. Baroni, A., e Abreu, F.B.: A formal library for aiding metrics extraction. In: 4th International Workshop on Object-Oriented Re-Engineering at ECOOP'2003, Darmstadt, Germany. (2003)
29. Goulão, M., e Abre, F.B.: Formalizing metrics for cots. In: Proceedings of the International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC'04) at the ICSE'2004, Edinburgh, Scotland (2004)
30. OMG: Object management group. OMG unified modeling language specification version 2.0 (2006)
31. McQuillan, J., Power, J.: Towards re-usable metric definitions at the meta-level. In: PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP), Nantes, France (2006)
32. Löwe, W., Panas, T.: Rapid construction of software comprehension tools. International Journal of Software Engineering and Knowledge Engineering **15**(6) (2005) 905–1023
33. Lincke, R., Löwe, W.: Validation of a standard- and metric-based software quality model. In: Proceedings of the 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2006). (2006) 81–90
34. Lincke, R., Löwe, W.: Compendium of software quality standards and metrics. http://www.arisa.se/compendium/ (2005)
35. Crescenzi, P., Kann, V., Karpinski, M., Woeginger, G.: A compendium of np optimization problems. http://www.nada.kth.se/~viggo/wwwcompendium/ (2006)