MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family

Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, Tudor Girba

MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family

(MSE et FAMIX 3.0 : un format d'échange de modèles et une famille de modèles de code)

Deliverable: 2.2 - Cutter ANR 2010 BLAN 0219 02

Stéphane Ducasse, Nicolas Anquetil,
Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, Tudor Girba

30 November 2011

Abstract

Software systems exceeding a certain critical size easily become difficult to maintain and adapt. Requirements change, platforms change and if a system does not evolve properly, its usefulness will decay over time. This document presents MSE a robust, scalable, extensible interexchange format and FAMIX 3.0 a family of metamodels to represent source code.

This deliverable is available as a free download.

First Edition, January, 2009. Final Edition, March, 2010.

Deliverable: 2.2

Title: MSE and FAMIX: an Interexchange format and source code model family

Titre: MSE et FAMIX : un format d'échange de modèles et une famille de modèles de code

Version: 1.0

**Authors: Stéphane Ducasse, Jannik Laval, Nicolas Anquetil,
Usman Bhatti, Andre Cavalcante-Hora, Tudor Girba**

Document identification

- First Public Version: 30 November 2011

Contents

Chapter1. Tools for Reengineering

Software systems exceeding a certain critical size easily become difficult to maintain and adapt. Requirements change, platforms change and if a system does not evolve properly, its usefulness will decay over time [DDN02]. Reengineering large industrial software systems is impossible without appropriate tool support [DT03].There is the scalability issue (millions of lines of code are the norm rather than the exception) and there is also the extra complexity of supporting and combining multiple tools with a wide variety of tasks (standard forward engineering techniques must be combined with additional reverse- and reengineering skills). The need for tool support in reengineering is reflected by the numerous tools or environments available in the reengineering research community [BG97, DT03].

Tools are not monolithic and are often based on multiple tools performing specific tasks: extracting information from a language, performing some analysis, displaying, computing reports.... Therefore there is a need for a specification of an *interexchange format* and also a *common source code meta model* to represent facts about software under analysis [DDT99, LTP04].

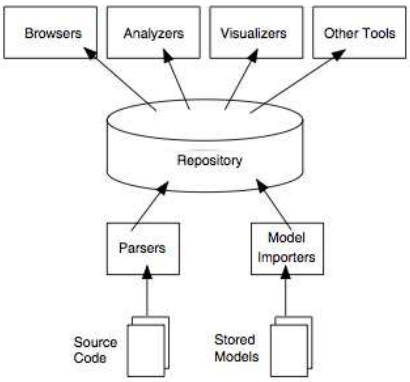

Figure 1.1: Typical tool infrastructure.

Figure 1.1 shows the general structure of a reengineering environment. At the lowest level it shows the source code or stored models which can be imported into the reengineering environment by appropriated parsers. The middle level shows the repository, which contains an abstracted model of the source code. The top level shows the tools that use the repository as their information base. The repository's meta-model is the central part that lets everything work together. The properties of the repository, and thus of the complete environment, are highly influenced by the meta-model that describes what and in which way information is modeled. The meta-model not only determines if the right information is available to perform the intended reengineering

tasks, but also influences issues such as scalability, extensibility, support for multiple models, and information exchange. The interexchange format is important because it serves as a glue between tools and model extractors often written in different languages.

Goals. In this deliverable we describe

- MSE a compact, simple and robust exchange format.
- FAMIX30 the third version of a common source code metamodel used to extract fact from languages in a language independent manner.

Chapter2. InterExchange Format

Several interexchange formats exist: Rigi standard format [Won98], CDIF [Com94, NTD98], TA [Hol98], GXL [HWS00], MSE used by the Moose, data exchange in Bauhaus [CEK+00]. Several attempts to unify them failed [Let98], GXL [HWS00]. In this section, we present MSE (for the moose interexchange format). MSE is based on the experience gained since 1996 in the development of the Moose software analysis platform [NDG05]: we started using CDIF (for ER based code models) [Com94], then XMI [XMI05, Gro98, Fre00, Sch01] (for EMOF based code models), and finally we designed MSE to be simple, scalable, robust (for FAME [KV08] based metamodels). MSE has been used to save FAMIX models [DTD01].

Some criteria. Repository information is often stored in text files. It is a lightweight way of storage, which is particularly well-suited for information exchange. Important criteria of the format which is used to store the information are human readability, lightweight manipulation, the ability to incrementally load information, and the non-influence of the entity loading order [LLL00]. Saint-Denis et al. [SDSK00] describe a set of 13 criteria that are important for an exchange format: Transparency (encoding and decoding processes specified by the model interchange format do not lose, add or alter any information contained in the original model), Scalability, Simplicity, Neutrality (model interchange format is independent of user-specific modeling constructs in order to allow a maximum number of model users to share model information), Formality (well specified), Flexibility, Evolvavility, Popularity, Meta-Model Identity, Solution Reuse, Legibility (ease with which a human may read and understand the format, facilitates manual manipulations on model interchange object), and Integrity (interchanged information reaches its destination without errors). Ducasse and Tichelaar present aspects that describe the actual structure of the exchange format because the structure has an impact on the functionality of the environment such as supporting multi-models or incremental loading [DT03]: flat-nested-chunk and support for nary.

2.1 MSE: a compact, simple and robust format

The MSE format allows one to specify models for import and export with Fame a simple meta meta model [KV08]. Similar to XML, MSE is generic and can specify any kind of data, regardless of the metamodel. In addition MSE is simple, compact, readable and extensible.

It is similar to XML, the main difference being that instead of using verbose tags, it makes use of parentheses to denote the beginning and ending of an element. MSE is based on the experience gained since 1996 in the development of the Moose software analysis platform [NDG05]. MSE has been used to save FAMIX models [DTD01].

2.1.1 An Example

The following snippet provides an example of a small model:
An example MSE file might look as follows

```
("Sample MSE file"
 (LIB.Library
    (librarian
       (LIB.Person
          (name 'Adrian Kuhn')))
    (books
       (LIB.Book
          (title 'Design Patterns')
          (authors (ref: 1) (ref: 2) (ref: 3) (ref: 4)))
       (LIB.Book
          (title 'Eclipse: Principles, Patterns, and Plug-Ins')
          (authors (ref: 1) (ref: 5)))
       (LIB.Book
          (title 'Smalltalk Best Practice Patterns')
          (authors (ref: 5)))))
 (LIB.Person (id: 1)
    (name 'Erich Gamma'))
 (LIB.Person (id: 2)
    (name 'Richard Helm'))
 (LIB.Person(id: 3)
    (name 'Ralph Johnson'))
 (LIB.Person(id: 4)
    (name 'John Vlissides'))
 (LIB.Person(id: 5)
    (name 'Kent Beck')))
```

The above MSE file describes a library with a librarian and 3 books by 5 authors. As you can see, it is either possible to nest elements (see the librarian) or the refer to them using the ref: tag.

2.1.2 For Meta models

In MSE we can also express the meta model of the model expressed above. The meta-model (or schema, if you prefer) of the file can be stored in the same format.

```
("Meta-model of above file"
 (FM3.Package
    (name 'LIB')
    (classes
       (FM3.Class
          (name 'Library')
          (attributes
             (FM3.Property
                (name 'librarian')
                (type (ref: 2)))
             (FM3.Property
                (name 'books')
                (multivalued true)
                (type (ref: 1)))))
       (FM3.Class (id: 1)
          (name 'Book')
          (attributes
             (FM3.Property
                (name 'title')
                (type (ref: String)))
             (FM3.Property
```

```
            (name 'authors')
            (multivalued true)
            (type (ref: 2)))))
      (FM3.Class (id: 2)
         (name 'Person')
         (attributes
            (FM3.Property
               (name 'name')
               (type (ref: String)))))))
```

2.1.3 For Models of Code

Here is an example of MSE used to exchange model of code.

```
( (FAMIX.Namespace (id: 1)
      (name 'aNamespace'))
  (FAMIX.Package (id: 201)
      (name 'aPackage'))
  (FAMIX.Package (id: 202)
      (name 'anotherPackage')
      (parentPackage (ref: 201)))
  (FAMIX.Class (id: 2)
      (name 'ClassA')
      (container (ref: 1))
      (parentPackage (ref: 201)))
  (FAMIX.Method
      (name 'methodA1')
      (signature 'methodA1()')
      (parentType (ref: 2))
      (LOC 2))
  (FAMIX.Attribute
      (name 'attributeA1')
      (parentType (ref: 2)))
  (FAMIX.Class (id: 3)
      (name 'ClassB')
      (container (ref: 1))
      (parentPackage (ref: 202)))
  (FAMIX.Inheritance
      (subclass (ref: 3))
      (superclass (ref: 2))))
```

The file defines 8 entities: 1 Namespace, 2 Packages, 2 Classes, 1 Method, 1 Attribute and 1 Inheritance. For each of these entities it provides a unique identifier (e.g., (id: 1)) and it defines properties. In general, properties can be either primitive, like (name 'aNamespace'), or they can point to another entity, like in the case of (container (ref: 1)) which denotes that the container property of ClassA points to the instance of Namespace named aNamespace.

The overall object graph can be seen graphically below (see Figure2.1).

2.2 MSE Specification

MSE is a file format to store FAME compliant metamodels and models [KV08]. Remark that all MSE files must use UTF-8 encoding.

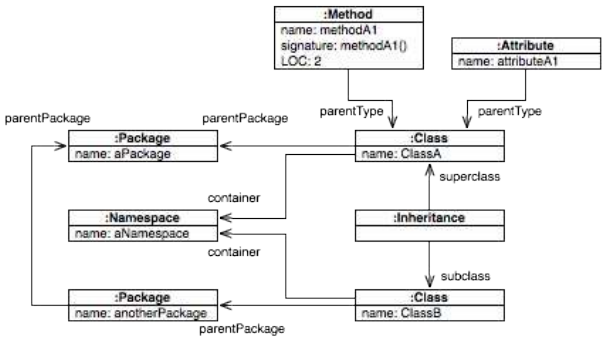

Figure 2.1: The object graph of the sample MSE file.

2.2.1 Grammar

The following grammar expressed in EBNF cover all the syntax of the MSE format where

- ? : which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one times).
- * : which means that something can be repeated any number of times (and possibly be skipped altogether).
- + : which means that something can appear one or more times.

The following

```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN ELEMENTNAME Serial ? AttributeNode * CLOSE
Serial := OPEN ID INTEGER CLOSE
AttributeNode := OPEN SIMPLENAME ValueNode * CLOSE
ValueNode := Primitive | Reference | ElementNode
Primitive := STRING | NUMBER | Boolean
Boolean := TRUE | FALSE
Reference := IntegerReference | NameReference
IntegerReference := OPEN REF INTEGER CLOSE
NameReference := OPEN REF ELEMENTNAME CLOSE

OPEN := "("
CLOSE := ")"
ID := "id:"
REF := "ref:"
TRUE := "true"
FALSE := "false"
ELEMENTNAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) * )
SIMPLENAME := letter ( letter | digit ) *
INTEGER := digit +
NUMBER := "-" ? digit + ( "." digit + ) ? ( ( "e" | "E" ) ( "-" | "+" ) ? digit + ) ?
STRING := ( "" [^] * "" ) +
```

```
digit := [0-9]
letter := [a-zA-Z_]
comment := """ [^"] * """
```

2.3 Conclusion

MSE represents our experience in exchanging source code models. Our experience with cumbersome CDIFs, verbose XMI formats led us to design a simple, readable, compact, incremental and robust interexchange format.

There is a missing part in MSE: the metadata about the extraction such as what is the system extracted, its date of extraction, the version and the tools used to extract it. Such information does not have to have a syntactic support per se and can be represented as special model entities.

Chapter3. Source Code Metamodels

Once we have a format defined, it is important to specify the model to represent source code elements. This section and the subsequent ones describe the global structure of the FAMIX model. They introduce the core model (which illustrates the core entities and associations), the abstract part of the model (defining the abstract superclasses that will be extended). This document describes the version 3.0 of the FAMIX metamodel. It is a major revision over FAMIX2.1 [DTD01].

FAMIX is a family of meta-models for representing models related to various facets of software systems. These meta-models are typically geared towards enabling analysis and they provide a rich API that can be used for querying and navigation. In this document we will not present the complete API but explain the key classes.

In addition it is important to understand that the goal of FAMIX is to represent in a language independent fashion programs. While this is possible to represent several languages with a common subsets, it is important to realize that

1. We have to do some compromises - sometimes for a given language a simpler solution could be better.

2. Meta model elements should be extended to support specific language features. In Moose we use class extensions (the fact that a method can be packaged in a different package than its classes, and that a class can get extra method when other packages are loaded) to extend the core metamodel classes without being forced to subclasses them.

3. While a meta model can be language independent, the analyses built on top of it may have to be language specific.

3.1 Overview of FAMIX

FAMIX is a family of meta models: it was extended to support history analysis [GD06, GDL04, DGF04, Gô5], aspects [FKDD11], duplication, coevolving entities [GDAK+07], files, authors, svn, [DGK06]. At it core it consists of a language independent meta-model that can represent in a uniform way multiple object-oriented and procedural languages. Figure 3.1 offers a an overview of the class hierarchy, including some of the most used extensions. The hierarchy is to be read from left to right, the entities to the right being specializations of those to the left.

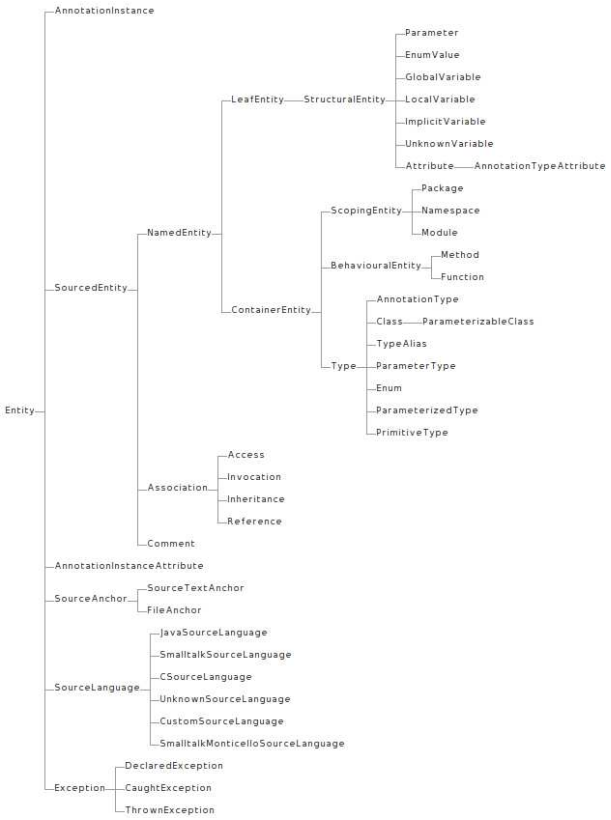

Figure 3.1: FAMIX core overview.

3.2 Core

In most cases, you get enough information if you master the core types entities that model an object-oriented system. These are Namespace, Package, Class, Method, Attribute, and the relationships between them, namely Inheritance, Access and Invocation. Figure 3.2 below provides an overview of these classes.

This model is an incorrect overview from two points of view:

1. It does not show all entities. For example, a Method has also Parameters and LocalVariables.

2. At places, it shows direct relationships when in reality they happen through inheritance. For example, the Access points to an Attribute, while in reality it points to a superclass (StructuralEntity). However, this picture is also useful because for most practical purposes it is all you need. Let us go through it step by step.

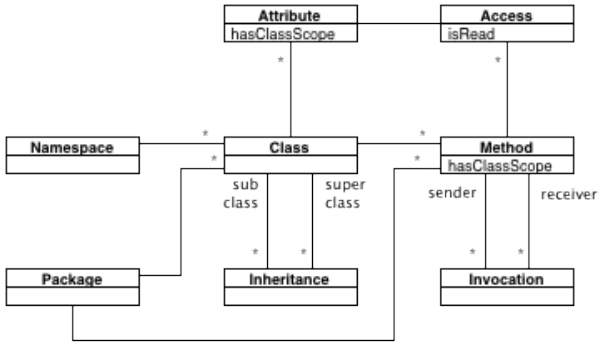

Figure 3.2: FAMIX core essential entities.

First, please note the missing arrows on the associations. This is not a mistake. In FAMIX, all associations are bidirectional.

Note that package and namespaces are two different entities. Indeed a package represents a group of entities but without providing a scoping mechanism. A namespace provides a scoping mechanism. This distinction is important because some languages only have namespaces (classes are grouped into namespaces), other only have one namespace and several packages.

Types.

Types are central to object-oriented systems. In this section, we take a closer look at the type hierarchy and related classes.

The root of the hierarchy is simply Type. This is a generic class representing a type in an object-oriented language. It can have many Methods and Attributes.

A type can also take part in inheritance relationships. This happens by means of Inheritance entities that connect pairs of types. Multiple inheritance is modeled by simply having multiple inheritance objects connecting the same subclass with multiple superclasses.

Type has several specializations for specific kinds of types. The most prominent is provided by Class. This models a typical class in Smalltalk, Java or C++, but it also models a Java interface (by means of the isInterface boolean property).

A PrimitiveType is just that: a primitive type. For example, int or char will be modeled using PrimitiveType entities.

ParameterizedType and ParameterizableClass model Java generics or C++ templates. In particular, a ParameterizableClass represents the generic definition, while the ParameterizedType represents the actual usage of the generic in a specific context.

Let me provide an example based on the following Java snippet:

```
public class ClassA<B,C> ...
...
public ClassA<ActualTypeA,ActualTypeB> anAttribute;
```

In this case, ClassA will be represented by a ParameterizableClass, and the declared type of anAttribute will be an actual ParameterizedType linking to ClassA. Fur-

thermore, B and C will be ParameterTypes, and the corresponding slots from the ParameterizedType will point to the actual types ActualTypeA and ActualTypeB.

3.3 Package Name: Famix-Core

Famix core is the general meta-model independent of any programming language. It should not contain language-specific features such as class extensions (Smalltalk), partial classes (C#), etc. Its root entity is Famix.Entity.

FAMIX.Entity extends Moose.Entity

FAMIXEntity is the abstract root class of the FAMIX meta-model entities.

Fields.

/annotationInstances: AnnotationInstance* → annotatedEntity

FAMIX.AnnotationInstance extends FAMIX.Entity

AnnotationInstance is an instance of an AnnotationType. It links an AnnotationType to an actual entity.

For example, the following is an annotation instance in Smalltalk. $<$primitive: 'primAnyBitFromTo' module:'LargeIntegers'$>$.

And the following is an AnnotationInstance in Java: @Test(timeout = 500)

Fields.

annotationType: AnnotationType → instances
annotatedEntity: Entity → annotationInstances
/attributes: AnnotationInstanceAttribute* → parentAnnotationInstance

FAMIX.AnnotationInstanceAttribute extends FAMIX.Entity

This models the actual value of an attribute in an AnnotationInstance.

In the following AnnotationInstance of Java, timeout is an annotation instance attribute : @Test(timeout = 500)

Fields.

parentAnnotationInstance: AnnotationInstance → attributes
annotationTypeAttribute: AnnotationTypeAttribute → annotationAttributeInstances
value: String

FAMIX.SourceAnchor extends FAMIX.Entity

FAMIXSourceAnchor is an abstract class representing a pointer to a source. The source can be identified in multiple ways as specified by the subclasses. The famix entity that this class is a source pointer for, is accessible via element property.

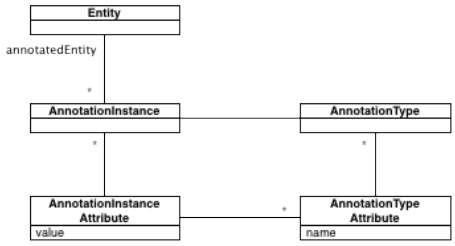

Figure 3.3: Annotations.

Fields.

element: SourcedEntity → sourceAnchor

FAMIX.SourceLanguage extends FAMIX.Entity

FAMIXSourceLanguage represents the programming language in which an entity is written. It is used for dispatching entity actions for specific languages. For example, formatting a source text will be performed according to the language. A project may have multiple languages.

A source language has a name and entities that are written in this language. One can create a default source language for a project by not associating any entities to it. In this case, all entities that do not have specific source langauge, belong to the default source language. One can attach entities to a sourceLanguage using addSourceEntity:.

Fields.

/name: String
/sourcedEntities: SourcedEntity* → declaredSourceLanguage

FAMIX.SourcedEntity extends FAMIX.Entity

FAMIXSourcedEntity models any fact in a program source and it is the superclass (root class) of all source code entities and their relationships. It is a FAMIXEntity and it can hold comments, a source anchor and a source language.

Fields.

sourceAnchor: SourceAnchor → element
declaredSourceLanguage: SourceLanguage → sourcedEntities
/comments: Comment* → container

FAMIX.CustomSourceLanguage extends FAMIX.SourceLanguage

FAMIXCustomSourceLanguage represents any source language that is not supported by default in moose. So, the CustomSourceLanguage is simply a possibility to specify some language from an outside parser without for which there are no specific tools defined. Actually, it just represents the name of the language with a string.

Fields.

name: String

FAMIX.UnknownSourceLanguage extends FAMIX.SourceLanguage

FAMIXUnknownSourceLanguage represents source language that has not been specified by the user.

The difference with CustomSourceLanguage is that people can export from outside a CustomSourceLanguage with a string representing the language, while the UnknownSourceLanguage is provided by default (null object pattern).

FAMIX.Association extends FAMIX.SourcedEntity

FAMIXAssociation is an abstract superclass for relationships between Famix named entities. It defines a polymorphic API refined by subclasses: essentially from, to, next and previous.

From and To properties are abstract at this level, but specific implementations can have multiple ends and properties. For example, FAMIXInheritance has: (i) From mapped to the subclass; (ii) To mapped to the superclass.

Next and Previous properties provide an order of the appearence of these associations in code. The order is calculated within a particular relationship for example, method invocation order within a calling method (from). For example in java, the following code method a() b(); c(); will produce two invocation associations first from method a to method b, and second from method a to method c. These associations are bound together and can be navigated with previous and next.

Fields.

previous: Association → next
/from: NamedEntity
/to: NamedEntity
/next: Association → previous

FAMIX.Comment extends FAMIX.SourcedEntity

FAMIXComment represents one instance of a comment (in the sense of programming language comments) for any Famix sourced entity. The commented sourced entity is called the container in the FAMIX model.

Fields.

content: String
container: SourcedEntity → comments

FAMIX.NamedEntity extends FAMIX.SourcedEntity

FAMIXNamedEntity is an abstract class, root of the hierarchy modeling source code entities. FAMIXNamedEntity has a name and it is physically present in source code. For example, methods, variables, types, namespaces. The name of the entity only contains the basic name and not the "fully qualified name". Apart from the name, it also has modifiers (e.g. public, protected, final, etc.) and it can be marked as a stub. A stub is a FAMIXNamedEntity that is used in the source code but its source is not available.

When applicable, a FAMIXNamedEntity also points to its containing package accessible via parentPackage.

Any of its subclasses must define the meaning of the belongsTo property, an abstract property that provides polymorphic traversal. For example, FAMIXClass defines belongsTo as being the container, while the FAMIXMethod defines belongsTo to point to the parentType. belongsTo can be used to calculate the "full qualified name" of a named entity. belongsTo is a derived property, which means that it is always computed from the information of other properties.

It can also return the list of invocations performed on this entity (considered as the receiver) (receivingInvocations).

Fields.

/isPublic: Boolean
/receivingInvocations: Invocation* → receiver
/isPrivate: Boolean
/isPackage: Boolean
/belongsTo: ContainerEntity
/isFinal: Boolean
/isProtected: Boolean
/nameLength: Number
name: String
/isAbstract: Boolean

modifiers: String*
parentPackage: Package → childNamedEntities
isStub: Boolean

FAMIX.Access extends FAMIX.Association

FAMIXAccess represents an access to a variable by a behavioural entity (for example, a function or a method).

For example if the method foo accesses the instance variable x, there is an access with the following information: (i) From: aFAMIXMethod (foo) (ii) To: aFAMIXAttribute (x)

aFAMIXMethod (foo) can be accessed using the message accessor (and from) aFAMIXAttribute (x) can be accessed using the message variable (and to).

Furthermore it can be tagged as read or write using isWrite: aBoolean.

For each access in the source code, there is one famix access created even if it is from the same behavioral entity towards the same variable.

Fields.

accessor: BehaviouralEntity → accesses
variable: StructuralEntity → incomingAccesses
/isRead: Boolean
isWrite: Boolean

FAMIX.Inheritance extends FAMIX.Association

FAMIXInheritance represents an inheritance relationship between one subtype (e.g. a subclass) and one supertype (e.g. a superclass).

To represent multiple inheritance, multiple instances of FAMIXInheritance should be created. FAMIXInheritance puts in relation two types, this way inheritance, for example, between classes and between interfaces can be modelled.

Fields.

subclass: Type → superInheritances
superclass: Type → subInheritances

FAMIX.Invocation extends FAMIX.Association

FAMIXInvocation represents the invocation of a message (signature) on a receiver by a behavioural entity. FAMIXInvocation has: (i) sender: the behavioral entity that sends the message; (ii) receiver: the structural entity

(variable) that receives the message; (iii) candidates: the list of potential behavioral entities that are actually being invoked. This is particularly useful for dynamic languages.

In an invocation, From is the sender of the message and To is the list of candidates. For each invocation in the source code, there is one famix invocation created even if it is from the same behavioral entity towards the same variable and the same message. For example in smalltalk, the following code anObject aSelector. will produce one invocation association from current method to a variable anObject with candidate aSelector. The list of candidates will also contain all the methods defining a similar signature as aSelector.

Fields.

candidates: BehaviouralEntity* → incomingInvocations
sender: BehaviouralEntity → outgoingInvocations
receiver: NamedEntity → receivingInvocations
signature: String
receiverSourceCode: String

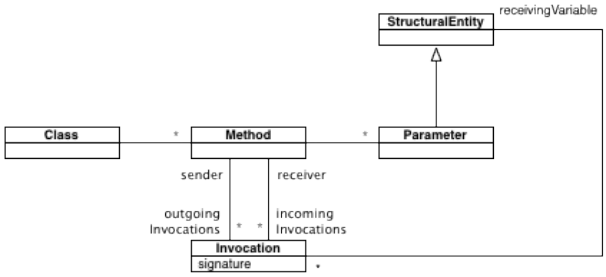

Figure 3.4: Invocations.

FAMIX.Reference extends FAMIX.Association

A FAMIXReference entity is created whenever one manipulates a class name as a variable. For example: (i) if the class is passed as a parameter to a method, or (ii) if a static method is invoked on a class.

For example, in the following Java code method a() B bObject = new B(); B.aStaticMethod(); There is only one reference which is created when the static method aStaticMethod is invoked on class variable B. In the declaration of B objects, the class B is the type of variable b but not a FAMIXReference. And instantiation new B() is an invocation of the default constructor, and not a FAMIXReference.

Note that FAMIXReference are defined between two FAMIXContainerEntity entities. So, it can also be used to represent dependencies between container entities that are computed from the dependencies of contained entities. For example, references between two packages can be computed from dependencies between classes of the packages.

Fields.

target: ContainerEntity → incomingReferences
source: ContainerEntity → outgoingReferences

FAMIX.ContainerEntity extends FAMIX.NamedEntity

FAMIXContainerEntity is the abstract superclass for source code entities containing other entities. Types, methods, and packages are examples of FAMIXContainerEntity.

Fields.

/incomingReferences: Reference* → target
(Java)/definedAnnotationTypes: AnnotationType* → container
/types: Type* → container
/outgoingReferences: Reference* → source

FAMIX.LeafEntity extends FAMIX.NamedEntity

FAMIXLeafEntity is the abstract superclass for source code entities that do not have children in Abstract syntax tree. For example, it represents variables of programming languages.

FAMIX.BehaviouralEntity extends FAMIX.ContainerEntity

FAMIXBehaviouralEntity is an abstract superclass for any kind of behavior. For example, functions and methods. It has a name because it is a named entity but it also has a signature in the format: methodName(paramType1, paramType2). The signature property is necessary for a behavioral entity. An external parser should provide a few metrics that cannot be derived from the model such as cyclomatic complexity, numberOfStatements and numberOfConditionals. Other metrics can be computed from the model if enough information is provided such as numberOfLinesOfCode (from source anchor) and numberOfComments (from FAMIXComment).

It provides properties to manage: (i) parameters (ii) local variables (iii) accesses to variables, and (iv) invocations to and from other behavioural entities.

Optionally, it can also specify a declaredType (e.g. return types for functions). This is useful for modeling behaviours from statically typed languages.

Fields.

/numberOfMessageSends: Number
numberOfStatements: Number
/numberOfAccesses: Number
/incomingInvocations: Invocation* → candidates
/localVariables: LocalVariable* → parentBehaviouralEntity
/accesses: Access* → accessor
cyclomaticComplexity: Number
numberOfLinesOfCode: Number
numberOfComments: Number
/outgoingInvocations: Invocation* → sender
numberOfConditionals: Number
/numberOfOutgoingInvocations: Number
numberOfParameters: Number
declaredType: Type → behavioursWithDeclaredType
signature: String
/parameters: Parameter* → parentBehaviouralEntity

FAMIX.ScopingEntity extends FAMIX.ContainerEntity

FAMIXScopingEntity represents an entity defining a scope at a global level.

Packages and Namespaces are two different concept in terms of scoping entity. Namespaces have semantic meaning in the language so they influence the unique name of the entity, while Packages are physical entities for packaging. In Smalltalk the two are explicitly different. In C++ we have explicit Namespaces, but not so explicit Packages. In Java, we have both Namespace (what you define in your Java source), and Package (the folder structure), but they happen to overlap in naming (although one is with . and the other one is with /) so people tend to see them as packages only.

Fields.

/childScopes: ScopingEntity* → parentScope
/functions: Function* → parentScope
/globalVariables: GlobalVariable* → parentModule
parentScope: ScopingEntity → childScopes

FAMIX.Type extends FAMIX.ContainerEntity

FAMIXType is a generic class representing a type. It has several specializations for specific kinds of types, the typical one being FAMIXClass. A type is defined in a container (instance of FAMIXContainer). The container is typically a namespace (instance of FAMIXNamespace), but may also be a class (in the case of nested classes), or a method (in the case of anonymous classes).

A type can have multiple subtypes or supertypes. These are modelled by means of FAMIXInheritance instances.

Fields.

/numberOfConstructorMethods: Number
/numberOfAttributesInherited: Number
/numberOfMethodProtocols: Number
/numberOfStatements: Number
/superInheritances: Inheritance* → subclass
/fanOut: Number
/methods: Method* → parentType
/weightOfAClass: Number
/totalNumberOfChildren: Number
/numberOfPublicMethods: Number
/typeAliases: TypeAlias* → aliasedType
/weightedMethodCount: Number
/hierarchyNestingLevel: Number
/numberOfComments: Number
/numberOfMethodsInHierarchy: Number
/numberOfAttributes: Number
container: ContainerEntity → types
/attributes: Attribute* → parentType
/numberOfRevealedAttributes: Number
/numberOfAnnotationInstances: Number
/subclassHierarchyDepth: Number
/numberOfPublicAttributes: Number
/behavioursWithDeclaredType: BehaviouralEntity* → declaredType
/numberOfAccessesToForeignData: Number
/fanIn: Number
/subInheritances: Inheritance* → superclass
/numberOfAbstractMethods: Number
/numberOfProtectedAttributes: Number
/numberOfMethodsInherited: Number
/numberOfMethodsOverriden: Number
/numberOfMethodsAdded: Number
/numberOfMethods: Number

/numberOfAccessorMethods: Number
/numberOfPrivateAttributes: Number
/numberOfProtectedMethods: Number
/numberOfMessageSends: Number
/structuresWithDeclaredType: StructuralEntity* → declaredType
/numberOfChildren: Number
/numberOfParents: Number
/numberOfPrivateMethods: Number
/tightClassCohesion: Number
/isAbstract: Boolean
/numberOfDuplicatedLinesOfCodeInternally: Number
/numberOfLinesOfCode: Number
/numberOfDirectSubclasses: Number

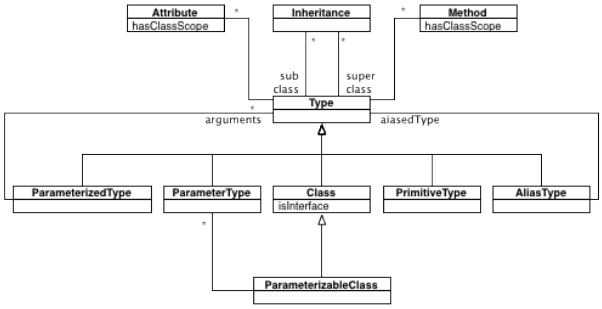

Figure 3.5: Types.

FAMIX.StructuralEntity extends FAMIX.LeafEntity

FAMIXStructuralEntity is the abstract superclass for basic data structure in the source code. A structural entity has a declaredType that points to the type of the variable.

Fields.

declaredType: Type → structuresWithDeclaredType
/incomingAccesses: Access* → variable

FAMIX.Function extends FAMIX.BehaviouralEntity

FAMIXFunction represents a behavioural entity in a procedural language.

Fields.

parentScope: ScopingEntity → functions
parentModule: Module

FAMIX.Method extends FAMIX.BehaviouralEntity

FAMIXMethod represents a behaviour in an object-oriented language.

A FAMIXMethod is always contained in a parentType.

Fields.

/isSetter: Boolean
(Java)/caughtExceptions: CaughtException* → definingMethod
/isConstructor: Boolean
(Java)/thrownExceptions: ThrownException* → definingMethod
/numberOfInvokedMethods: Number
hasClassScope: Boolean
/numberOfAnnotationInstances: Number
/isGetter: Boolean
/hierarchyNestingLevel: Number
timeStamp: String
/isConstant: Boolean
(Java)/declaredExceptions: DeclaredException* → definingMethod
parentType: Type → methods
kind: String
/isOverriden: Boolean
/isOverriding: Boolean
/isInternalImplementation: Boolean

FAMIX.Namespace extends FAMIX.ScopingEntity

FAMIXNamespace represents a namespace from the source language. Namespaces have semantic meaning in the language so they influence the unique name of the entity.

A namespace denotes an entity that has meaning from a language point of view. For example, in C++, there exist a concept with the same name that has no other responsibility beside providing a lexical scope for the contained classes and funcions.

When an entity is placed inside a namespace, the fully qualified name (mooseName) is affected.

Fields.

numberOfAttributes: Number
/numberOfClasses: Number
/instability: Number
/numberOfNonInterfacesClasses: Number
/numberOfMethods: Number
/distance: Number
/bunchCohesion: Number
/afferentCoupling: Number
/efferentCoupling: Number
/numberOfLinesOfCode: Number
/abstractness: Number

FAMIX.Package extends FAMIX.ScopingEntity

FAMIXPackage represents a package in the source language, meaning that it provides a means to group entities without any baring on lexical scoping.

Java extractors map Java packages to FAMIXNamespaces. They can also mirror the same information in terms of FAMIXPackage instances.

Fields.

/numberOfProviderPackages: Number
/weightedMethodCount: Number
/bunchCohesion: Number
/numberOfClasses: Number
/distance: Number
numberOfClientPackages: Number
/numberOfOutportClasses: Number
/instability: Number
numberOfMethods: Number
/afferentCoupling: Number
/childNamedEntities: NamedEntity* → parentPackage
/relativeImportanceForSystem: Number
/efferentCoupling: Number
/totalNumberOfLinesOfCode: Number
/numberOfHiddenClasses: Number
/abstractness: Number

FAMIX.AnnotationType extends FAMIX.Type

FAMIXAnnotationType represents the type of an annotation. In some languages, Java and C#, an annotation as an explicit type. An AnnotationType can have a container in which it resides.

Fields.

container: ContainerEntity → definedAnnotationTypes
/instances: AnnotationInstance* → annotationType

FAMIX.Class extends FAMIX.Type

FAMIXClass represents an entity which can build new instances. A FAMIXClass is a FAMIXType, therefore it is involved in super/sub types relationships (depending on the language) and it holds attributes, methods.

FAMIX does not model explicitly interfaces, but a FAMIXClass can represent a Java interface by setting the isInterface property.

A class is typically scoped in a namespace. To model nested or anonymous classes, extractors can set the container of classes to classes or methods, respectively.

Fields.

/numberOfInternalDuplications: Number
isInterface: Boolean
/numberOfExternalDuplications: Number

FAMIX.PrimitiveType extends FAMIX.Type

It represents a primitive type. For example, int or char are modeled using PrimitiveType entities. Void is also considered a primitive type.

FAMIX.TypeAlias extends FAMIX.Type

This entity models a typedef in C.

Fields.

aliasedType: Type → typeAliases

FAMIX.Attribute extends FAMIX.StructuralEntity

FAMIXAttribute represents a field of a class. It is an attribute of the parent type.

Fields.

/numberOfGlobalAccesses: Number
parentType: Type → attributes

/numberOfAccessingClasses: Number
hasClassScope: Boolean
/numberOfAccesses: Number
/hierarchyNestingLevel: Number
/numberOfAccessingMethods: Number
/numberOfLocalAccesses: Number

FAMIX.GlobalVariable extends FAMIX.StructuralEntity

FAMIXGlobalVariable represents a global variable in the source code.

Fields.

parentScope: ScopingEntity
parentModule: Module → globalVariables

FAMIX.ImplicitVariable extends FAMIX.StructuralEntity

FAMIXImplicitVariable represents a variable defined by the compiler in a context, such as self, super, thisContext.

Fields.

parentBehaviouralEntity: BehaviouralEntity

FAMIX.LocalVariable extends FAMIX.StructuralEntity

FAMIXLocalVariable represents a local variable in the scope of a behavioural entity.

Fields.

parentBehaviouralEntity: BehaviouralEntity → localVariables

FAMIX.Parameter extends FAMIX.StructuralEntity

FAMIXParameter represents one parameter in a method declaration.

Fields.

parentBehaviouralEntity: BehaviouralEntity → parameters

FAMIX.UnknownVariable extends FAMIX.StructuralEntity

FAMIXUnknownVariable represents some unknown entity encountered while importing the project, possibly due to a syntax error in the source code.

FAMIX.AnnotationTypeAttribute extends FAMIX.Attribute

This models the attribute defined in a Java AnnotationType. In Java, annotation type attributes have specific syntax and use.

For example, in Java the following AnnotationType has four AnnotationTypeAttributes (id, synopsis, engineer and date are attributes).

public @interface MyAnno int id(); String synopsis(); String engineer() default "[unassigned]"; String date() default "[unimplemented]";

When using an annotation, an annotation instance is created that points back to the annotation type. The annotation instance has attributes that are annontation instance attributes and point back to their annotation type attributes.

Fields.

/annotationAttributeInstances: AnnotationInstanceAttribute* → annotationTypeAttribute

/parentAnnotationType: AnnotationType

3.4 Package Name: Famix-SourceAnchor

Famix source anchor adds to the famix core meta-model representation of access to the source code of the famix entities.

FAMIX.FileAnchor extends FAMIX.SourceAnchor

This offers a source anchor that connects a sourced entity to a file through a relative path stored in the fileName. In addition, the source can be further specified with a startLine and an endLine number in the file.

Fields.

endLine: Number
startLine: Number
startColumn: Number
fileName: String
endColumn: Number

FAMIX.SourceTextAnchor extends FAMIX.SourceAnchor

This stores the source as an actual string variable. It is to be used when it is not possible to link to the actual source.

3.5 Package Name: Famix-Java

Famix java adds to the famix core meta-model representation of language specific features for Java language describing entities such as exceptions, generics, and enums.

FAMIX.Exception extends FAMIX.Entity

This is the abstract representation of an Exception. It is specific to Java. It points to an exceptionClass. The class of a FAMIXException is a normal FAMIXClass.

Fields.

exceptionClass: Class

FAMIX.CaughtException extends FAMIX.Exception

This is an exception that is explicitly handled by a method. For example, in Java it is an exception that appears in a catch statement.

Fields.

definingMethod: Method → caughtExceptions

FAMIX.DeclaredException extends FAMIX.Exception

This is an exception explicitly declared as being thrown by a method.

Fields.

definingMethod: Method → declaredExceptions

FAMIX.ThrownException extends FAMIX.Exception

This is an exception explicitly thrown by a method.

Fields.

definingMethod: Method → thrownExceptions

FAMIX.JavaSourceLanguage extends FAMIX.SourceLanguage

FAMIXJavaSourceLanguage represents the Java programming language in which an entity is written.

FAMIX.Enum extends FAMIX.Type

This models an enum in Java.

For example, the following Java code defines an Enum with seven EnumValues.

public enum Day SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

Fields.

/values: EnumValue* → parentEnum

FAMIX.ParameterType extends FAMIX.Type

ParameterType represents the symbolic type used in parameterizable classes. This is a FAMIXType.

Example: public class AClass$<$A,B,C> ...

Where AClass is a ParameterizableClass. A, B and C are ParameterType of AClass.

FAMIX.ParameterizedType extends FAMIX.Type

FAMIXParameterizedType represents a type with arguments. So, it is an instantiation for the use of FAMIXParameterizableClass. It can appear as a type of an attribute, a type of a local variable, a parameter of a method, a return of a method, etc. Example:

... public Map$<$String,Collection> anAttribute; ...

Where Map$<$String,Collection> is the FAMIXParameterizedType of anAttribute. String and Collection are arguments. Map is the parameterizableClass.

Fields.

arguments: Type*
parameterizableClass: ParameterizableClass

FAMIX.EnumValue extends FAMIX.StructuralEntity

It models the values defined in an FAMIXEnum. These are attributes of enums with default values.

For example, the following Java code defines an Enum with seven EnumValues.

public enum Day SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

Fields.

parentEnum: Enum → values

FAMIX.ParameterizableClass extends FAMIX.Class

ParameterizableClass represents the definition of a generic class with parameters. The parameters of the entity are modeled as ParameterType.

Example: public class AClass$<$A,B,C> ...

Where AClass is a ParameterizableClass. A, B and C are ParameterType of AClass.

Fields.

/parameters: ParameterType*

3.6 Package Name: Famix-File

Famix file adds to the famix core meta-model representation of a filesystem.

FILE.AbstractFile extends FAMIX.Entity

The abstract file system class. It is subclassed by File and Folder.

Fields.

name: String

FILE.File extends FILE.AbstractFile

It represents a file in the file system.

Fields.

numberOfInternalMultiplications: Number
totalNumberOfLinesOfText: Number
/numberOfExternalDuplications: Number
numberOfCharacters: Number
numberOfEmptyLinesOfText: Number
numberOfLinesOfText: Number
numberOfExternalClones: Number
numberOfInternalClones: Number
numberOfDuplicatedFiles: Number
averageNumberOfCharactersPerLine: Number
/numberOfInternalDuplications: Number
numberOfKiloBytes: Number
numberOfBytes: Number

FILE.Folder extends FILE.AbstractFile

It represents a folder in the file system. It can contain other files or folders.

Fields.

numberOfFolders: Number
totalNumberOfLinesOfText: Number
numberOfLinesOfText: Number
numberOfFiles: Number
numberOfEmptyLinesOfText: Number

3.7 Package Name: Famix-C

Famix C adds to the famix core meta-model representation of language specific features for C language describing entities such as module.

FAMIX.CSourceLanguage extends FAMIX.SourceLanguage

FAMIXCSourceLanguage represents the C language.

FAMIX.Module extends FAMIX.ScopingEntity

FAMIXModule represents a that basically provides a simple scoping abstraction for a .C/.CPP/.H files.

3.8 Conclusion

This document presents a simple, compact, readable interexchange format. Then it presents the core of the FAMIX a family of metamodels.

Bibliography

[BG97] Berndt Bellay and Harald Gall. A comparison of four reverse engineering tools. In *Proceedings of WCRE (Working Conference on Reverse Engineering)*, pages 2–11. IEEE Computer Society Press: Los Alamitos CA, 1997.

[CEK+00] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner. Data exchange in Bauhaus. In *Proceedings WCRE '00*. IEEE Computer Society Press, November 2000.

[Com94] CDIF Technical Committee. CDIF framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, January 1994. See http://www.cdif.org/.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings of the International Conference on The Unified Modeling Language (UML'99)*, volume 1723 of *LNCS*, pages 630–644, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[DGF04] Stéphane Ducasse, Tudor Gîrba, and Jean-Marie Favre. Modeling software evolution by treating history as a first class entity. In *Proceedings Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 75–86, Amsterdam, 2004. Elsevier.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DT03] Stéphane Ducasse and Sander Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 15(5):345–373, October 2003.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[FKDD11] Johan Fabry, Andy Kellens, Simon Denier, and Stéphane Ducasse. Aspectmaps: A scalable visualization of join point shadows. In *International Conference on Program Comprehension (ICPC)*, pages 121–130. IEEE Computer Society Press, 2011.

[Fre00] Michael Freidig. XMI for FAMIX. Informatikprojekt, University of Bern, June 2000.

[GÎ05] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[GDAK+07] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Daniel Raţiu. Using concept analysis to detect co-change patterns. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 83–89. ACM Press, 2007.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, September 2004. IEEE Computer Society.

[Gro98] Object Management Group. XML Metadata Interchange (XMI). Technical Report ad/98-10-05, Object Management Group, February 1998.

[Hol98] Richard C. Holt. An introduction to TA: The Tuple-Attribute language. Technical report, University of Waterloo, November 1998.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE '00*, November 2000.

[KV08] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

[Let98] Timothy C. Lethbridge. Requirements and proposal for a Software Information Exchange Format (SIEF) standard. Technical report, University of Ottawa, November 1998. http://www.site.uottawa.ca/~tcl/papers/sief/standardProposalv1.html.

[LLL00] Sébastien Lapierre, Bruno Laguë, and Charles Leduc. Datrix(tm) source code model and its interchange format: Lessons learned and considerations for future work. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[NTD98] Oscar Nierstrasz, Sander Tichelaar, and Serge Demeyer. CDIF as the interchange format between reengineering tools. In *OOPSLA '98 Workshop on Model Engineering, Methods and Tools Integration with CDIF*, October 1998.

[Sch01] Andreas Schlapbach. Generic XMI support for the MOOSE reengineering environment. Informatikprojekt, University of Bern, June 2001.

[SDSK00] Guy Saint-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format. the SPOOL case study. In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, 2000.

[Won98] Kenny Wong. The rigi user's manual — version 5.4.4. Technical report, University of Victoria, 1998.

[XMI05] Xml metadata interchange (xmi), v2.0, 2005. http://www.omg.org/cgi-bin/doc?formal/05-05-01.