Electronic Communications of the EASST
Volume X (2007)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

A Meta-model for expressing first-class changes

Peter Ebraert, Bart Depoortere and Theo D'Hondt

10 pages

Guest Editors: Tom Mens, Maja D'Hondt, Kim Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Meta-model for expressing first-class changes

Peter Ebraert[1], **Bart Depoortere**[1] **and Theo D'Hondt**[1]

[1] pebraert, bdepoort, tjdhondt@vub.ac.be, http://prog.vub.ac.be/
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium

Abstract: First-class changes were proven to provide useful information about the evolution history of software programs. The subjects of first-class changes are expressed on the building blocks of the program which they affect. Those building-blocks are described by a meta-model. The goal of this paper is to find a proper *meta-model* to express *first-class changes*. We first establish four criteria for comparing different meta-models and form the basis of a taxonomy to classify them. Afterwards, some meta-models are evaluated with respect to those criteria. Famix is found as the best match with respect to the imposed criteria. Famix, however, still has some minor shortcomings, which can be overcome by the extensions we propose to it.

Keywords: Meta-model, software evolution,

1 Introduction

First-class changes are objects which represent change and can be referenced, queried and passed along [EPD07]. They were proven to provide useful information about the evolution history of software programs [RL07]. The subject of a first-class change object is expressed on the building blocks of the program which they affect. Those building-blocks are described by a meta-model. The goal of this paper is to find a *meta-model* suitable to express *first-class changes*.

The remainder of this paper is structured as follows. In Section 2, we establish four criteria to which the meta-models need to adhere in order to be express first-class changes. Afterwards, we use the established criteria in Section 3 to compare some existing meta-models and discuss about them. Section 4 discusses about the meta-model which is found to be the best suited for modeling changes and proposes some extensions to that meta-model. We conclude in Section 5 and provide some avenues of future work.

2 Criteria for meta-models

In order to compare the different meta-models, we first have to identify the criteria which are important with respect to modeling changes. This section introduces four criteria which are relevant with respect to modeling changes on software programs.

Support for multiple programming languages – It is desirable to obtain a *cross-language specification of change types*. Such a specification could be used as an intermediary format for software evolution tools to study, compare and exchange information about the evolution of applications which were possibly implemented in different programming languages. This would bring along the extra advantages that the software evolution tools do not need to be specified for each programming language separately, but rather on the cross-language specification itself.

A cross-language specification of change types can only be achieved if the meta-model of their building blocks supports *multiple programming languages*. To accomplish this, the meta-model must be as general as possible and thus omit language specific features.

Extensibility hooks – The expressiveness of artifacts is often to limited in order to reason thoroughly about evolution facets of a software application. Especially when language-specific features are omitted in the meta-model to support a cross-language specification. This can be overcome by allow the core of the meta-model to be *extended*. Extending the meta-model, however, may decrease the degree of language independence (e.g. extending it to be able to cope with language specific features). As such, a good balance between a cross-language core and language specific extension possibilities must be established.

Derivable system invariants – Design inconsistencies should be avoided at all times. Applying changes on the design of an application, however, influence the design and could introduce inconsistencies. In order to avoid that, *system invariants* – constraints that must be satisfied by the system at any time – can be of great help. All different change types are annotated with pre- and post-coniditions which verify wether an instantiation of that change type does not violate the system invariants.

The system invariants are actually imposed by the meta-model. While some meta-models provide an extensive list of the invariants, others only provide them implicitly. In both cases, we require that the meta-model serves as a basis for deriving those invariants.

Easy information exchange – The *cross-language specification of change types* may be used by the software evolution tools as an intermediary format to communicate. Therefore that specification and the meta-model upon which it is based must be complete, consistent, easy to interpret and can not contain any ambiguities. These properties are further referred to as *easy information exchange*. A high degree of language independence increases the ease of information exchange while a high degree of extensibility decreases the ease of information exchange. Therefore it is important to find a good balance between language independence and extensibility.

3 Towards a taxonomy of meta-models

This section discusses four alternative meta-models and their support with respect to the previously defined criteria.

3.1 The Unified Modeling Language (UML)

UML is a general-purpose modeling language widely used in the world of software engineering [BJR96, Gro99]. It includes a graphical notation used to specify, visualize, construct and document designs. The UML specification consists of a *meta-model* that describes the language for specifying UML models (e.g. class diagrams).

Support for multiple programming languages – UML supports the entire software development process starting from analysis and design omitting implementation specific issues. Therefore its meta-model is designed to model software systems implemented in various classed-based object-oriented programming languages and thus supports multiple languages.

Extensibility hooks – The UML meta-model provides three extensibility mechanisms. *Tagged values* permit users to annotate any model element with extra information (value) paired with a keyword (tag). Figure 1 shows an example of a tagged value in the bottum right corner. It is visualized as a note on the design while the keyword `documentation` is stored in the background.

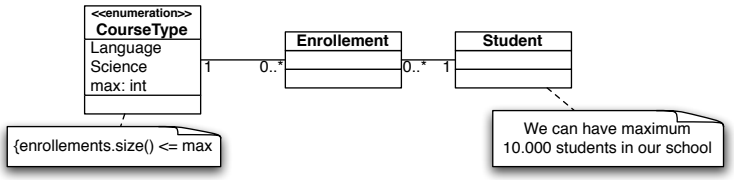

Figure 1: UML tagged values, stereotypes and constraints - Example

Stereotypes allow users to further specialize model elements, it is an extensibility mechanism equivalent to inheritance. Figure 1 depicts an example of a stereotype: the `enumeration` stereotype denotes that `CourseTypes` is an enumeration providing some defined types of courses.

Users can apply semantic restrictions to model elements by using *constraints* which may be specified in free-form text or in *Object Constraint Language* (OCL). OCL is a declarative language that is part of the UML standard and is used for describing rules that apply to UML models. An example of OCL is expressed in the left bottum corner of Figure 1. Students have the possibility to enroll for courses from a certain type whereas each course type has a maximum number of allowed students.

Derivable system invariants – System invariants can be derived from the meta-model's elements. They can be derived for instance, from the relationships between those entities, their cardinalities or OCL constraints.

Easy information exchange – UML is very expressive due to its large number of available concepts and its extensibility mechanisms. This, however, makes it harder to exchange information about UML models. Furthermore, there are some additional problems associated with the UML meta-model specification which decrease the ease of information exchange. First, it is incomplete, vague and inconsistent [HS01, RW99]. Second, modelers using OCL restrict their audience since OCL is a complex language and few people can read and write it [Amb04]. As such, we conclude that UML does not provide an easy information exchange.

3.2 RevJava

RevJava is a tool which operates on compiled Java code and checks if that software systems is conform to specified design rules [Flo02]. Its meta-model defines all relevant concepts of a Java software system (e.g. package, class or method) and the associations between them (e.g. inheritance definition, method call or variable access).

Support for multiple programming languages – the meta-model is designed for modeling Java-programs but it is general enough to capture the core concepts of software systems implemented in other class-based object-oriented programming languages.

Extensibility hooks – RevJava's meta-model is incorporated in the RevJava tool. The author has not explicitly specified any extensibility mechanisms in the document describing the tool.

Derivable system invariants – system invariants can be derived from the Java meta-model specification. All derived invariants however apply to the Java programming language and only in some extent to other class-based programming languages.

Easy information exchange – RevJava's meta-model is easy to read and understand since its number of available concepts remains small. No extensibility mechanisms are specified by RevJava. As such, we can conclude that it provides support for an easy information exchange.

3.3 FAMIX

FAMIX stands for *FAMOOS Information Exchange Model* and was created to support information exchange between interacting software analysis tools by capturing the common features of different object-oriented programming languages needed for software re-engineering activities [DTS99, DD99, Tic01].

Figure 2 shows a conceptual view of the FAMIX model. On the left side, we see different programming languages used to implement several case studies. On the right side, we see various experiments conducted by several software analysis tools on the provided case studies. In the middle, we see the information exchange model that only captures the common features of class-based object-oriented programming languages such as classes or methods. To cope with language specific features, the FAMIX model can be extended by using the provided hooks represented by the grey bars at the bottom of the figure.

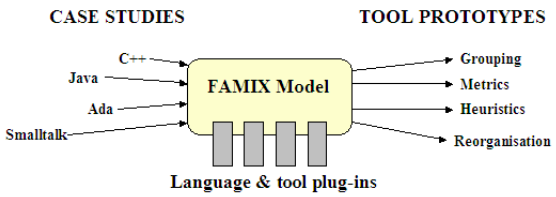

Figure 2: Conception of the FAMIX model (based on [DTS99])

Support for multiple programming languages – the FAMIX model is designed to model software systems at source code-level independent from the implementation language. To achieve language independence, the FAMIX model only captures common features of different class-based object-oriented programming languages, omitting language-specific features. It thus supports multiple languages.

Extensibility hooks – The FAMIX model provides three extensibility mechanisms. *New concepts* can be defined in order to specify new model elements. *New attributes* can be added to existing concepts in order to store additional information in the model elements. *Annotations* can be added by the user to any model element for attaching extra information to it.

Derivable system invariants – System invariants can be derived from the model's elements. They can be derived for instance, from the relationships between those entities, their cardinalities or their constraints.

Easy information exchange – The FAMIX model was created to *support information exchange between tools*. Hence, it provides very good support for an easy information exchange.

3.4 Graph based alternative

Mens and Lanza suggest representing software systems as graphs. Program entities are then represented by nodes, relationships between them by edges [ML02]. To accomplish that, they specified a *typed meta-model* consisting of typed edges (e.g. inheritance and accesses) and typed nodes (e.g. class or method). Multiple edges between two nodes are allowed and attributes can be added to each node or edge.

Support for multiple programming languages – The meta-model behind the graph representation supports any programming language whose concepts can be represented by either nodes or edges.

Extensibility hooks – The authors have not specified any extensibility mechanisms. Extensibility seems possible but is definitely not eased by the provided meta-model.

Derivable system invariants – System invariants can be derived from the meta-model's specification.

Easy information exchange – The used meta-model is easy to read and understand since its number of available concepts remains small. Extensibility is not encouraged. This helps in providing an easy information exchange.

3.5 Comparison of alternatives

Table 1 shows an overview in which the stated criteria are compared against the different alternatives discussed in the previous sections. An "X" indicates that the corresponding criterium is badly or not supported by the meta-model while a "V" indicates the opposite. The figure shows that the FAMIX model scores the best of all explored alternatives as it supports all criteria.

Requirements/ Meta-Model	UML	RevJava	FAMIX	Graph
Support for multiple languages	V	V/X	V	V
Extensibility hooks	V	X	V	X
Derivable system invariants	V	V	V	V
Easy information exchange	X	V	V	V

Table 1: Comparison of alternatives

4 Extensions to FAMIX

Some software systems may never be shut down and require modifications to their source-code at run-time. In those cases, it may be useful to capture the dynamic information of that running system (e.g. the creation of a new instance). We propose to extend the FAMIX model with notions that capture the state of a running system.

Any object-oriented programming language (e.g. Smalltalk or Java) defines entities unique to that language. The second extension we propose copes with language specific artifacts in order to derive Smalltalk specific changes (e.g. FAMIX multiple inheritance vs Smalltalk single inheritance). The following two subsections respectively explore the extensions for dynamic state and Smalltalk-specific features.

4.1 Extension for dynamic state

The FAMIX model does not provide any elements to store dynamic information such as living instances of a particular class or the value of some global variable. It is however necessary to store dynamic information whenever one wants to specify changes concerning the dynamic software evolution. An example could be a garbage collecting functionality that removes all non-referenced instances. The following subsections discuss the dynamic extension of the FAMIX model.

Instance – The green parts in Figure 3 show the extensions regarding living instances of a certain class. A new class `Instance` (inheriting from the `Object` class) has been added which

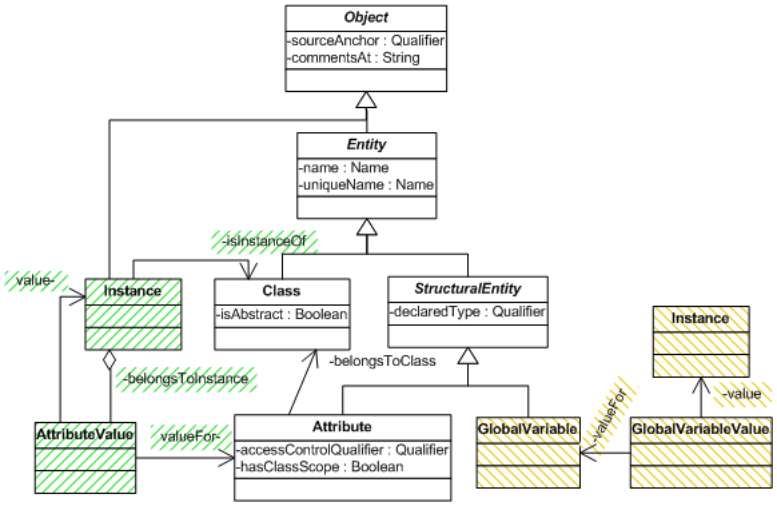

Figure 3: Dynamic extension - Instance & Global Variable

keeps a reference to the class of which it is an instance (denoted by the `isInstanceOf` relationship). This enables an `Instance` instance to query the referenced `isInstanceOf` class for all its defined attributes and methods. `AttributeValue` has been added to hold the value of a particular `Attribute` belonging to an `Instance` instance (`belongsToInstance`).

Global variable – The orange parts of Figure 3 show the extensions regarding adding global variables of any running program. `GlobalVariableValue` keeps a reference to the global variable of which it holds the value (`valueFor`) which in its turn is an `Instance` instance.

4.2 Extension for Smalltalk

The FAMIX model serves as a meta-model for different implementation languages without specifying language specific artifacts. As such, it does not cover the Smalltalk-specific language features. When expressing change types about Smalltalk programs, hoever, such extensions are desirable. This section deals with this kind of extensions and is based on the work of Tichelaar who extended the FAMIX model to capture Smalltalk's language features [Tic01]. Extra modifications are provided to capture information not suggested by Tichelaar. Following subsections discuss these extensions: extensions suggested by Tichelaar are indicated with "(T)", additional extensions are denoted by "(*)".

Class – Each Smalltalk class has an associated metaclass that describes it. This metaclass does not have its own name hence Smalltalk generates a name by concatenating the base class name with the " class" `String`. FAMIX defines a `Class` class allowing to model both class types. The orange parts in Figure 4 shows that one attribute has been added: `isMetaClass`, a `Boolean` indicating whether or not the class represents a Smalltalk metaclass.

Figure 4: Smalltalk extension - Class & Behavioral entity

Behavioral entity – Return types of methods are not explicit in Smalltalk. Tichelaar proposes to populate `declaredReturnClass` and `declaredReturnType` with the most general type of an object-oriented programming language namely `Object`. Tichelaar states that functions are not used in Smalltalk which implies that the `Function` entity of FAMIX will never be populated [Tic01]. Smalltalk, however, does allow *block closures* which are first-class anonymous functions that take a number of arguments and have a body. In our extension, we use the `Function` entity to represent Smalltalk's block closures.

The green part in Figure 4 shows that the `BehavioralEntity` class was extended with an `inferredReturnClass` association which refers to all possible candidates for the return type of the concerned behavioral entity. Tichelaar has pushed down the following attributes to the `Method` entity. Each method has a unique `signature` and a `isPureAccessor`, which is a `Boolean` that indicates whether or not the represented method is a pure getter/setter.

Figure 5 also shows that the `belongsToProtocol` relationship has been added. A *protocol* is the name for a group of methods allowing to organize them. For instance the "accessing" protocol groups all accessing methods (getters and setters).

An association between the `Method` and `Package` classes has been added as a method belongs to exactly one package in Smalltalk. The value of the `belongsToPackage` reference may differ from the package in which the containing class is defined. The `isConstructor` field indicates whether or not the behavioral entity creates and initializes new instances of its containing class.

Structural entity – Smalltalk is a dynamically typed language meaning it does not require the developer to explicitly type variables. Type checking happens at run-time and types of variables are determined by the values assigned to them. Therefore Tichelaar proposes to populate the `declaredType` field and `declaredClass` association with the most general type of an object-oriented programming language: `Object`.

The green parts of Figure 5 reveal that the `inferredClass` association has been added to the `StructuralEntity` class. This association refers to all possible candidates for the type of the structural entity. Smalltalk allows initialization of attributes and global variables.

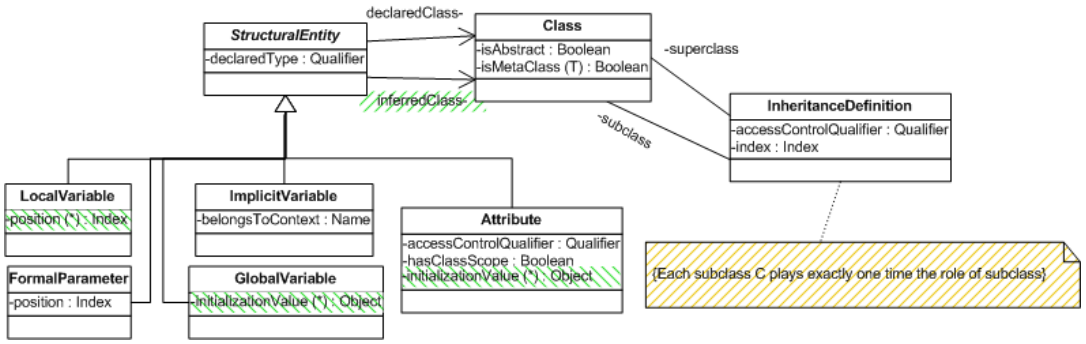

Figure 5: Smalltalk extension - Structural entity & Inheritance definition

That is why the `initializationValue` attribute has been added to the `Attribute` and `GlobalVariable` classes. Furthermore, the `position` attribute has been added to the `LocalVariable` class and maintains the index of the local variable in the behavioral entity's list of temporary variables. Smalltalk imposes that all attributes are protected and are only accessible within the defining class and its subclasses. Hence Tichelaar proposes to populate the `accessControlQualifier` attribute with the "protected" `Qualifier`.

Inheritance definition – The FAMIX model allows multiple inheritance whereas Smalltalk does not. In Smalltalk, classes always inherit from one single class (except the root class, `Object`). The red part in Figure 5 depicts a constraint imposing single inheritance. Tichelaar proposes to populate the index attribute of the `InheritanceDefinition` class with the `null` value since in this case an index has no meaning. Inheritance in Smalltalk is always publicly accessible: all methods (public) and attributes (protected) are inherited by the subclass and have the same visibility.

5 Conclusion and future work

A meta-model can be considered as an explicit description of which building blocks (program entities) are defined in the model of the programming language(s) adhering to it. A meta-model is needed to derive *first-class changes* which are suitable units that can express the evolution of a software system. This paper discusses four criteria for choosing the appropriate meta-model: *support for multiple programming languages*, *extensibility hooks*, *derivable system invariants* and *easy information exchange*. Four different meta-models were analyzed with respect to those four criteria. The analysis reveals that Famix satisfies all four criteria.

The FAMIX model serves as a *language independent meta-model* and was introduced to exchange information between different software analysis tools that study the architecture at *source-code level* of class-based object-oriented software. Two extensions to the FAMIX model are presented. The first extension involves the capability to express changes on the dynamic state of a running system. The second extension copes with language specific artifacts in order to

derive Smalltalk specific changes. The most important track of future work consists in testing whether the extended Famix meta-model is expressive enough to specify first-class change types on class-based object-oriented programming.

Acknowledgements: We want to thank the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen), for providing Peter Ebraert with a doctoral scholarship, and as such financing this research.

Bibliography

[Amb04] S. Ambler. *The Object Primer Third Edition Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2004.

[BJR96] G. Booch, I. Jacobson, J. Rumbaugh. The Unified Modelling Language for Object-Oriented Development. Documentation set, version 0.9, Rational Software Corporation, 1996.

[DD99] S. Ducasse, S. Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, 1999.

[DTS99] S. Demeyer, S. Tichelaar, P. Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, University of Berne, 1999.

[EPD07] P. Ebraert, E. V. Paesschen, T. D'Hondt. Change-Oriented Round-Trip Engineering. In *Atelier RIMEL: Rapport de recherche*. Volume VAL-RR2007-01. 2007.

[Flo02] G. Florijn. RevJava: Design critiques and architectural conformance checking for Java software. *SERC*, 2002.

[Gro99] O. M. Group. Unified Modeling Language 1.3. Technical report, Rational Software Corporation, June 1999.

[HS01] B. Henderson-Sellers. Some problems with the UML 1.3 meta-model. In *Proceedings of the 34th Hawaii International Conference on System Sciences*. IEEE Computer Society, 2001.

[ML02] T. Mens, M. Lanza. A Graph-Based Metamodel for Object-Oriented Software Metrics. *Electronic notes in Theoretical Computer Science* 72(2):12, 2002.

[RL07] R. Robbes, M. Lanza. A Change-based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science* 166:93–109, 2007.

[RW99] G. Reggio, R. Wieringa. Thirty one Problems in the Semantics of UML 1.3 Dynamics. 1999.

[Tic01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.