

EMF4CPP: a C++ Ecore Implementation

Andrés Senac

Dept. of Inf. & Comm. Eng.

Dept. Comp. Eng.

University of Murcia

30.071 Murcia, Spain

Diego Sevilla

Dept. Comp. Eng.

University of Murcia

30.071 Murcia, Spain

Gregorio Martínez

Dept. of Inf. & Comm. Eng.

University of Murcia

30.071 Murcia, Spain

Abstract

Eclipse, and its Ecore metamodel, have become a *de facto* standard for providing a complete working environment for model-driven development. However, being Java-centric, there is no established mapping from Ecore to the C++ language, partly because of the difficulty in overcoming the differences between these two programming languages in areas such as type mapping, memory management, and run-time capabilities. EMF4CPP fills this gap by providing a C++ implementation and type mapping for the Ecore metamodel. It offers a C++ code generator with a consistent type mapping, a reader and a serializer for XMI-encoded models and metamodels, being able to be used in model-to-model transformations, and establishing the basis for developing a set of tools for MDD in C++ that offer efficient construction, management, and usage of models and metamodels. The paper introduces the mapping, discusses performance issues and shows motivating usage scenarios. Preliminary tests show C++ code to manipulate models to be 1.5 times faster than the corresponding Java code, and using 35% less memory, also favoring its usage in embedded environments.

1 Introduction

The Eclipse Project [3] and its EMF (*Eclipse Modeling Framework*) [15] have contributed to new and encouraging developments in the area of *Model-Driven Development* (MDD) in gen-

eral, and to *Metamodeling* in particular.

The EMF approach, with the Ecore metamodel, aimed at providing a simpler-yet rich enough-model elements description that allowed, among other things, (1) describing model artifacts characteristics (such as classes and attributes) in a platform- and language-independent way, and (2) to map those model elements to implementation technologies and programming languages constructs.

Models conforming to the Ecore metamodel have a direct mapping to the Java language constructs. However, there is no established mapping to the C++ programming language, limiting the interoperability between the powerful set of tools available around the EMF technology and *Model-Driven Developments* (MDD) wanting to leverage libraries and techniques available in C++ (such as *metaprogramming* [6] and *Embedded Domain-Specific Languages* (DSL) [7].) to offer alternative tools and abstractions for *text-to-model* (T2M), *model-to-model* (M2M), and *model-to-text* (M2T) transformations.

EMF4CPP tries to fill this gap by being able to generate C++ implementation artifacts from Ecore-based metamodels. The generated code artifacts support the whole Ecore defined characteristics, including classes, attributes, operations, references, etc., as well as being able to read and write model and metamodel data using the XMI model interchange format [13], offering a reflective mapping to work with models and metamodels in a generic way, and including a consistent mapping of Ecore types to C++ types.

Thus, the main advantages of EMF4CPP can be summarized as follows:

- There is no standard C++ type mapping for the Ecore metamodel. EMF4CPP can be seen as a first step to standardize this mapping. This, not only has the direct benefit of being able to use the different EMF tools (and Ecore metamodels) to describe the data model of C++ applications, but also it represents a basis to interoperate with future MDD tools that use C++ as the programming language.
- It offers a reflective API for models and metamodels, being able to interoperate with other tools in intermediate model transformations (through XMI), and serving as a basis for tools that exploit C++ idioms and patterns for dealing with model creation and transformation. Also, projects that have to support several implementation languages can share metamodels, and steps can be done into automatic conversion of code between implementation languages.
- By providing a standard C++ mapping, we can easily embed scripting languages such as Python, Lua, or Ruby for writing tools that manage models and metamodels. We have already implemented preliminary support for Python (PyEcore).
- Finally, initial results (in Section 3) show C++ around 1.5 times faster and with 35% less memory footprint in managing model data structures. This is a key factor in embedded systems, that usually have strong memory and processing power constraints. Also, it represents a real-world advantage in dealing with bigger models (for instance, those resulting of using ADM/KDM [14].)

Along with these benefits, the development of EMF4CPP was motivated mainly by two industrial projects carried out under the Cátedra SAES-UMU [16]:

- **Generic Interface Testing:** We developed an abstract model for describing

interfaces defined using either CORBA IDL, Java, or C++, and tests performed on them, including both static (e.g. output values given input values, either obtained from previous execution logs or using mathematical functions), and dynamic (e.g. workflow-based) tests. A DSL was developed for specifying these tests. The test specification allowed us to automatically generate both testing clients *and* mocking servers. Finally, we also modelled the run-time behavior of the testing process, i.e. which tests have been performed, calls made, including input and output values, which tests failed or succeeded, etc. In the case, for example, of CORBA IDL/C++ clients, we needed a way of representing the modelled run-time tests state into C++ classes (as it can be done with the Java version using the EMF tools.) These classes were obtained using EMF4CPP.

- **Usability assessment and Validation on GUIs:** Another project for which we needed EMF4CPP was related to reasoning about Qt4-based GUIs [12], abstracting the properties and characteristics of Qt graphical widgets in order to do run-time usability assessment and validation of input and output values. The abstraction was actually performed by doing a model-to-model transformation of the original model of Qt widgets to the abstract model of widget properties needed for either usability or validation in GUIs. For instance, in some usability processes, we were interested in the containment relation of widgets, as well as relative positions, instead of the actual type of the widget (i.e. if a widget could be seen as a *container* widget.) Alternatively, for input validation, text boxes and spin boxes were treated just as widgets producing a value that could be checked for validity. The usability and validation algorithms were performed on the different abstract models for widgets. As Qt is based on C++, and as those algorithms are performed mostly at run-time, we needed a

way of representing, manipulating, and transforming models in C++ at run-time, hence the usage of EMF4CPP.

Other similar efforts include EMF4Net [2], a port of EMF to the .NET Platform, RMOF [8], an implementation of MOF and Ecore in Ruby, and RGen [11], a Ruby-based modeling and generator framework.

This paper is structured as follows: Section 2 describes the main concepts behind the Ecore metamodel and introduces the C++ implementation, and examples of usage compared to the Java counterpart. Section 3 shows some performance measurements showing the viability of the approach. Finally, Section 4 shows some conclusions, the current implementation status, and future work.

2 The EMF4CPP Design and Implementation

The Ecore metamodel defines the characteristics of several main abstractions, among the most important:

- **Classes:** Define the *class* concept and their characteristics, such as inheritance, class attributes, operations, and references to other classes.
- **References:** Specify the characteristics of dependencies between class' instances (for example, the physical containment property.)
- **DataTypes:** State the intrinsic characteristics of general data types. Common data types such as integer (EInt), double precision floating point (EFloat), etc., are provided.
- **Packages:** Define a name-space based containment architecture for data types, classes, and other packages.

As we stated above, Ecore also provides trivial mappings of all those data types and concepts to Java implementation artifacts. However, Ecore is general enough to allow mappings to other programming languages to be built.

The approach we followed when providing a C++ mapping for Ecore is depicted in Figure 1. In the current implementation, the source code generator was developed with EMF itself, particularly using Xpand and Xtend [4]. From the Ecore metamodel itself, it produces C++ source code that gets packed as a shared library. These classes allow us to manage, describe, and process any metamodel described using Ecore.

Figure 2 shows the process followed by any metamodel conforming to Ecore. The generator creates C++ classes that allow manipulating entities of the types defined in the metamodel, and also reading model instances (i.e. data to fill the classes defined in the metamodel) and process them within a C++ application, linking the correct dynamic libraries.

Thus, EMF4CPP consists of two parts: a source code generator from Ecore metamodels to C++ and two runtime support libraries. One of the runtime support libraries implements the Ecore metamodel (`libecore`). The other one allows to parse and serialize models in XMI format (`libecorecpp`).

2.1 Ecore to C++ Type Mappings

One of the most challenging issues in bringing the mapping of Ecore to C++ was to fill the gap between the type system and the memory management of Java and C++. Table 1 shows the mapping arranged for the Ecore data types.

The mapping comprises several parts: (1) mapping types and classes, (2) memory management, (3) the implementation of the reflective API (which is used for serialization to and from XMI documents), and (4) the change notification API.

Regarding the type mapping, we decided to map to the closest C++ type (for example EBoolean to `bool` and EClasses to C++ classes, EPackages to C++ namespaces, etc.), and ignore the differences between boxed (EBooleanObject) and non-boxed ones (EBoolean), because this distinction does not make sense in C++ since it does not have a supertype *Object*.

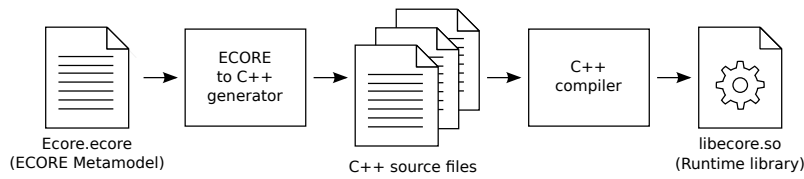


Figure 1: EMF4CPP implementation workflow.

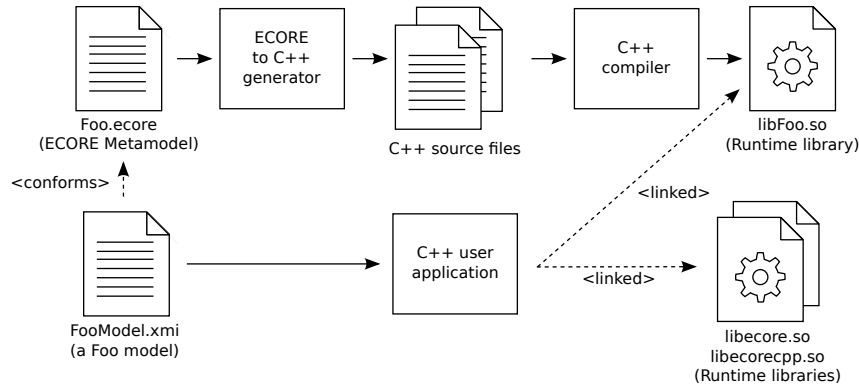


Figure 2: Generating C++ classes from an arbitrary metamodel.

The mapping of attributes and references of classes is as follows:

- For each attribute or reference P , accessor (`getP()`) and mutator (`setP()`) methods are provided. Unlike the Java mapping, accessor methods for attributes do not modify the object instance, so they usually return a *constant C++ reference* to the real attribute.
- If that attribute or reference is *multiple*, additional methods `addP()`, `setPAt()`, and `deletePAt()` are added to respectively add a new element to the list, set the n th element, and delete the n th element.
- A template method `as<type>()` is provided by the `EObject` class for explicit conversion between related types without the need of temporary variables.

For the memory management, we decided to use raw pointers to represent references. Thus, a class named “MyClass” in the metamodel

would be translated into both a class with the same name, and a type “MyClass_ptr”, with the semantics of a raw C++ pointer. Given the strict meaning of containment (ownership) *vs.* non-containment references in the Ecore metamodel, a class always knows which of its own structural features to *free* when an instance of that class gets reclaimed (as opposed to Java, where memory management is automatic.)

The reflective API allows accessing models using a generic API that does not depend on the actual types of the contained elements. This supposes a challenge, as C++ does not have a common object root, and its run-time reflective capabilities are minimal compared to those offered by Java. So, we implemented enough API to be able to query and modify models in a generic way, *but* those model classes must have been previously generated by the generator (conversely, Java allows creating instances of metamodels without previously having to generate model classes.) The

EDataType	C++ type	EDataType	C++ type
EBigDecimal	long double ^a	EBigInteger	long long ^a
EBoolean	bool ^b	EBooleanObject	bool ^b
EByte	unsigned char	EByteArray	std::vector<unsigned char>
EChar	char	ECharObject	char
EDate	time_t	EDiagnosticChain	(undefined)
EDouble	double	EDoubleObject	double
EEnumerator	int	EFeatureMap	(undefined)
EFeatureMapEntry	(undefined)	EFloat	float
EFloatObject	float	EInt	int
EIntObject	int	EJavaObject	any ^c
ELong	long	ELongObject	long
EResource	(undefined)	EResourceSet	(undefined)
EShort	short	EShortObject	short
EString	std::string ^d	EInvocationTargetException	(undefined)

^aThis may change to use, for example, types provided by the GMP library [5].

^bWe decided to use the same basic C++ type for boxed and non-boxed types for Java. (See Section 2.1.)

^cA type similar to the Boost library [1] `boost::any`.

^dThe current implementation also allows the usage of `std::wstring` and `wchar_t` as *character type*, for example, in Windows platforms.

Table 1: Ecore to C++ Type Mappings.

full reflective API is doable in C++, but left as a future work as it is not strictly needed to work with models and metamodels. Additionally, we mapped the EJavaObject type to an any type that can hold any C++ value.

Finally, the change management API is not implemented in this first version. It is left as a future work. An `_initialize()` method is provided to ensure the referential integrity of the model (see Figure 5.)

2.2 Model Usage and Comparison with Java

In this Section we show the EMF4CPP API usage creating a simple metamodel and a model that conforms to that metamodel. In this case, we are going to create the *company* metamodel shown in Figure 3 using our C++ Ecore runtime library. It contains three meta-classes (*Company*, *Department*, and *Employee*). This metamodel example is slightly modified from [10].

Figure 4 shows an excerpt of the creation of the *company* metamodel using our Ecore runtime library. In particular, the creation of the *Company* meta-class is shown, with the addition of the “name” attribute of type EString.

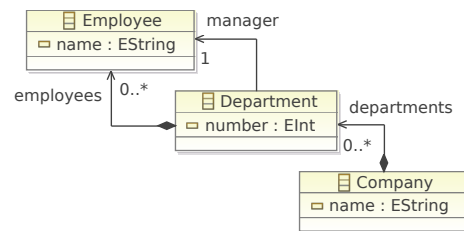


Figure 3: Example company metamodel (slightly modified from [10].)

The API follows the conventions seen in Section 2.1: the `setName()` method for setting the “name” property, and the `addEStructuralFeatures()` method to add a new structural feature to the property containing the list of them. Finally, the three new meta-classes are added to the *company* package.

Correspondingly, listing in Figure 5(a) shows the creation of a simple model conforming to the metamodel. The *Company* class (and its corresponding *Company_ptr* type) have been previously generated by the EMF4CPP generator. Note how all the dif-

```

using namespace ecore;

EcoreFactory_ptr ecoreFactory =
    EcoreFactory::_instance();
EcorePackage_ptr ecorePackage =
    EcorePackage::_instance();

// Create a Company class
EClass_ptr companyClass = ecoreFactory->
    createEClass();
companyClass->setName("Company");

// Add name attribute to a Company class
EAttribute_ptr companyName = ecoreFactory->
    createEAttribute();
companyName->setName("name");
companyName->setEType(ecorePackage->
    getEString());
companyClass->addEStructuralFeatures(
    companyName);

...

// Create a package that represents company
EPackage_ptr companyPackage = ecoreFactory
    ->createEPackage();
companyPackage->setName("company");
companyPackage->setNsPrefix("company");
companyPackage->setNsURI("http://com.
    example.company.ecore");
companyPackage->addEClassifiers(
    employeeClass);
companyPackage->addEClassifiers(
    departmentClass);
companyPackage->addEClassifiers(
    companyClass);

```

Figure 4: Excerpt. Company metamodel creation.

ferent attributes and relations (references) are correctly established. A last call to `_initialize()` completes the model to ensure the referential integrity (e.g. opposite references). Here model deletion is performed automatically using a `std::auto_ptr`.

Figure 5(b) shows the equivalent Java code for the example above. There is no need of the final initialization pass, as EMF implements change notification, left as a future work.

Figure 6 shows the usage of the reflective API to create a Company class instance, establishing its “name” attribute, then reading it, and finally obtaining the list of references to Departments. Note the usage of the `any` type and the special covariant list type `EEListBase`.

```

// Get the Classifier by name
EClass_ptr companyClass =
    companyPackage->getEClassifier("Company")
    ->as< EClass >();

// Create an instance
EObject_ptr company = companyFactory->
    create(companyClass);

// Get the name structural feature by name
EStructuralFeature_ptr companyName =
    companyClass->getEStructuralFeature("name");

// Set the name attribute
EString name("University of Murcia");
any a(&name);
company->eSet(companyName, a);

// Get the name attribute
a = company->eGet(companyName);
const EString* obtained_name = any::
    any_cast< const EString *>(a);
std::cout << *obtained_name << std::endl;

// Get the list of references to
// Departments
EStructuralFeature_ptr departments =
    companyClass->getEStructuralFeature("
    departments");
a = company->eGet(departments);
mapping::EEListBase_ptr dep_list =
    any::any_cast< mapping::EEListBase_ptr >(
    a);
// Obtain first dept.
EObject_ptr department = (*dep_list)[0];

```

Figure 6: Reflective API example.

3 Performance Comparison

3.1 Mapping-related Performance Considerations

Our first approach at memory management was using Boost `boost::shared_ptr` construct, as it manages the memory safely and automatically for pointers. However, this introduced performance tradeoffs, and was not usable in the case of circular references. As mentioned, using raw pointers and following the containment relationship for references was enough for a correct memory management.

Also, using `any` for the reflective API boxed all the features of classes, including data types and even list of references (those declared, for example, as `0..*`), which implied a memory copy of the whole list of references. So, we had to tweak the reflective API so that `any`s re-

```

using namespace company;

CompanyPackage_ptr companyPackage =
    CompanyPackage::_instance();
CompanyFactory_ptr companyFactory =
    CompanyFactory::_instance();

// Create a company
std::auto_ptr<Company> umu(
    companyFactory->createCompany());
umu->setName("University_of_Murcia");

// Create a department
Department_ptr catSAES =
    companyFactory->createDepartment();
catSAES->setNumber(8515);

// Create employees
Employee_ptr asenac =
    companyFactory->createEmployee();
asenac->setName("Andres_Senac");
catSAES->addEmployees(asenac);

Employee_ptr dsevilla =
    companyFactory->createEmployee();
dsevilla->setName("Diego_Sevilla");
catSAES->addEmployees(dsevilla);

// Set the department manager
catSAES->setManager(dsevilla);

// Add the department
umu->addDepartments(catSAES);

// Initialize the model
umu->_initialize();

// (model is deleted automatically)

```

(a) C++ code.

```

import company;

CompanyPackage companyPackage =
    CompanyPackage.eINSTANCE;
CompanyFactory companyFactory =
    CompanyFactory.eINSTANCE;

// Create a company
Company umu =
    companyFactory.createCompany();
umu.setName("University_of_Murcia");

// Create a department
Department catSAES =
    companyFactory.createDepartment();
catSAES.setNumber(8515);

// Create employees
Employee asenac =
    companyFactory.createEmployee();
asenac.setName("Andres_Senac");
catSAES.getEmployees().add(asenac);

Employee dsevilla =
    companyFactory.createEmployee();
dsevilla.setName("Diego_Sevilla");
catSAES.getEmployees().add(dsevilla);

// Set the department manager
catSAES.setManager(dsevilla);

// Add the department
umu.getDepartments().add(catSAES);

// Initialize the model
// (not needed)

// (model is deleted automatically)

```

(b) Java code.

Figure 5: Company model population.

turned *references* to the class elements. Also, unlike Java, C++ standard containers are not covariant with the generic type (for example, `std::list` is not a subclass of `std::list<A>` even if `B` is a subclass of `A`), so in case of having to obtain a set of references using the reflective API, we had to build a new list with generic `EObject` references. To overcome this penalty, we implemented a covariant list type exclusively for the reflective API.

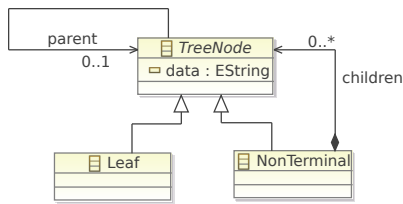
3.2 Performance Evaluation

To show the validity of the approach in the Ecore C++ implementation, we designed a performance evaluation test to compare it with the Eclipse Ecore implementation. The test

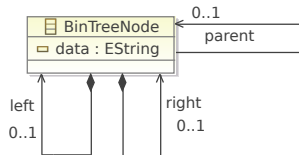
consisted in reading a model conforming to the simple tree metamodel in Figure 7(a) with a non-terminal root node with one million children leaves, doing a trivial model transformation to build a model conforming to the metamodel in Figure 7(b), and serializing the resulting model.

The transformation included simple model manipulations to measure the overhead of working with the model structure itself. In particular, it consisted on building an ordered non-balanced binary tree, lexicographically comparing the “data” attribute of the source model leaves. We implemented the same transformation in both Java and C++.

For the comparison we measured the time of loading the model (48MB of XML), model



(a) Source metamodel.



(b) Destination metamodel.

Figure 7: Metamodels used for performance testing.

transformation, and the time of writing the resulting metamodel (164MB), in a 3.0 GHz AMD Phenom II X4 940 with 8GB of RAM under 64-bit Linux, and in an Intel Core2Quad Q8300 with 4GB of RAM under 32-bit Linux. The C++ code was compiled using GCC 4.4 with -O3 optimization. The Java code was run under Sun JDK 1.6.0. We tested 32-bit and 64-bit because of the difference in pointers size, as model data structures are mainly pointer-based.

Table 2 shows the times obtained for both implementations. Model loading times are around 1.13 times faster in 32-bit systems, with a clear bigger speedup of 1.64 in 64-bit systems. Processing times vary from 1.6 (in 32-bit) to 2.37 (in 64-bit) times faster in C++ than in Java. Model writing times range between around 2 to 3.5 times faster in C++.

	EMF	EMF4CPP	Speedup
Load (32b)	3941	3503	1.13
Transf. (32b)	2920	1778	1.64
Serialization (32b)	3878	2011	1.93
Load (64b)	3696	2145	1.72
Transf. (64b)	2096	886	2.37
Serialization (64b)	4466	1286	3.47

Table 2: Times in milliseconds and speedup.

Table 3 shows the memory usage after load-

ing the source model and making the transformation. The memory usage in Java has been measured after calling the garbage collector. Savings of around 35% were obtained.

	EMF	EMF4CPP	Reduction
L&T (32b)	141,604,848	89,124,579	37.06 %
L&T (64b)	234,417,976	159,354,063	32.02 %

Table 3: Memory usage in bytes of load and transformation and percentage of reduction.

4 Conclusions, Current Status and Future Work

EMF4CPP provides a consistent and efficient Ecore to C++ mapping that can be used to follow a model-driven approach for C++ developments. The current implementation allows to generate C++ code from Ecore metamodels and to manage and use models conforming to that metamodels.

Thorough the paper we mentioned a set of features that were intended for future work, such as to complete the Ecore to C++ mapping, the fully reflective API support, and the implementation of the change management API.

Also, we plan to develop infrastructure tools to manage models, and provide T2M, M2M, and M2T transformations, leveraging the C++ metaprogramming techniques to the MDD paradigm. For example, as an alternative to the *text-to-model* workflows in Eclipse, we want to explore using the metaprogramming, the embedded DSL-based Spirit parser [7] and monadic parsers similar to Haskell Parsec [9] to automatically generate models from C++-defined parsers. We also want to explore bringing the mapping to C++-embeddable scripting programming languages such as Lua and Python. EMF4CPP is also being submitted to become an Eclipse Incubator project.¹

Another interesting area of future work is the specification of a higher level meta-

¹ <http://www.eclipse.org/forums/index.php?t=msg&goto=542311&>.

metamodel (of which Ecore would be an instance) that allows a more seamless language mappings (for example, Ecore specifies the EJavaObject type, too tied to Java, instead of a more generic one). It could also include element characteristics such persistence or remote access properties, allowing to model distributed systems and *Service Oriented Architectures* (SOA).

EMF4CPP has been developed by the Cátedra SAES Team as an Open Source contribution. Source code and additional information is available at <http://catedrasaes.inf.um.es/trac/wiki/EMF4CPP> under the LGPL License.

Acknowledgements

This paper has been partially supported by the Cátedra SAES of the University of Murcia [16], a joint effort between SAES (Sociedad Anónima de Electrónica Submarina, <http://www.electronica-submarina.com>) and the University of Murcia (<http://www.um.es>) to work on open-source software, and real-time and critical information systems.

References

- [1] Dawes, B., Abrahams, D., Rivera, R., et al.: Boost C++ Libraries (2010), <http://boost.org/> (last visited May 2010)
- [2] Eclipse Foundation: EMF4Net Proposal (2010), http://wiki.eclipse.org/EMF4Net_Proposal (last visited May 2010)
- [3] Eclipse Foundation: The Eclipse Project (2010), <http://www.eclipse.org>
- [4] Eclipse Foundation: Xpand and Xtend (2010), <http://www.eclipse.org/modeling/m2t/?project=xpand#xpand> (last visited May 2010)
- [5] Free Software Foundation: The GNU Multiple Precision Arithmetic Library (GMP) (2010), <http://gmplib.org/> (last visited May 2010)
- [6] Gurtovoy, A., Abrahams, D.: Boost MPL Library (2004), <http://www.boost.org/doc/libs/release/libs/mpl>
- [7] de Guzman, J., Kaiser, H.: The Spirit Parser (2009), <http://boost-spirit.com>
- [8] Jim Steel, Franck Fleurey, Jesús Sánchez Cuadrado: RMOF (2005), <http://rmof.rubyforge.org/> (last visited May 2010)
- [9] Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators for the Real World. Tech. Rep. UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht (2001), <http://legacy.cs.uu.nl/daan/parsec.html>
- [10] Litani, E., Merks, E., Steinberg, D.: Discover the Eclipse Modeling Framework (EMF) and Its Dynamic Capabilities (August 2005), <http://www.devx.com/Java/Article/29093/0/page/1>
- [11] Martin Thiede: RGen (2009), <http://ruby-gen.org> (last visited Jun 2010)
- [12] Nokia Corp.: Qt: A Cross-Platform Application and UI framework (2010), <http://qt.nokia.com/>
- [13] Object Management Group: XML Metadata Interchange (XMI), version 2.1.1 (2007), document formal/2007-12-01. <http://www.omg.org/spec/XMI/2.1.1/>
- [14] Object Management Group: Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), version 1.2 (2010), document formal/2010-06-03
- [15] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF, Eclipse Modeling Framework. Addison-Wesley, second edn. (2009)
- [16] University of Murcia, SAES: Cátedra SAES-UMU (2010), <http://www.um.es/catedrasaes>