An EMF-like UML Generator for C++

Sven Jäger, Ralph Maschotta, Tino Jungebloud, Alexander Wichmann, and Armin Zimmermann

Systems and Software Engineering Group
Computer Science and Automation Department
Technische Universität Ilmenau
Ilmenau, Germany
Contact: see http://www.tu-ilmenau.de/sse/

Keywords: Model Based Software Development, Code Generation, Meta Modeling, C++, UML, MOF, Ecore

Abstract: Model-driven architecture is a well-known approach for the development of complex software systems. The most famous tool chain is provided by Eclipse with the tools of the Eclipse modeling project. Like Eclipse itself, these tools are based on Java. However, there are numerous legacy software packages written in C++, which often use only an implicit meta-model. A real C++ implementation of this meta-model would be necessary instead to be used at run time. This paper presents a generator for C++ to create the classes, meta-model packages, and factories to realize modeling, transformation, validation, and comparison of UML models. It gives an overview of its workflow and major challenges. Moreover, a comparison between Java and C++ implementations is given, considering different benchmarks.

1 INTRODUCTION

The Model Driven Architecture (MDA) approach as defined by the Object Management Group (OMG) is a well-known family of standards which unifies every step of the development of an application from Platform-Independent Model (PIM) through Platform-Specific Models (PSM) to generated code and a deployable application (OMG, 2014). The common Eclipse Modeling Project (EMP) supports this approach by providing a unified set of modeling frameworks, tool support, and standard implementations based on the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009; Gronback, 2009). It uses Ecore, a meta meta-model which is similar to the essential meta object facility (EMOF) for describing a meta-model. The EMF framework with its code generation facility can be used to build tools and other applications based on a structured data model, like complete editors for Ecore-based domain specific meta-models. It produces Java classes for the model and other necessary utilities to create and edit such a model. Therefore, the MDA approach is well integrated with the Java programming language, but it is weakly supported for other programming languages such as C++.

C++ is one of the most commonly used programming languages (TIOBE Software BV, 2015). Numerous legacy software packages are written in C++ and use often only an implicit meta-model. To use a real meta-model at runtime, an implementation of the meta-model in C++ would be necessary (Jungebloud et al., 2013). Such a C++ meta-model could be used, for instance, to configure dialog properties at runtime, to realize a runtime Object Constraint Language (OCL) (OMG, 2012) for the checking of model elements, or to execute a behavior which is described by activity diagrams.

There are mainly two possibilities to create such meta-model representations. The obvious one is to write code and implement every class manually. When the number of elements in a model increases, however, the best way to overcome the implementation is to use a generator. A big advantage of a generator (besides saving implementation time) is the deterministic result of the transformation. Each model element is transformed in the same way and produces similar code blocks. By using guidelines like the MISRA C++ (MISRA, 2008) for the generator, the resulting code can be used in safety-critical software.

EMF4CPP (González et al., 2010) is an available implementation of a C++ Ecore meta-model and comes with a generator for Ecore to C++ transformation. Preliminary results show that "...memory consumption and efficiency is usually better in EMF4CPP than in Java..." (EMF4Cpp, 2015). Unfortunately, the last visible development activities in this project date back to the year 2011.

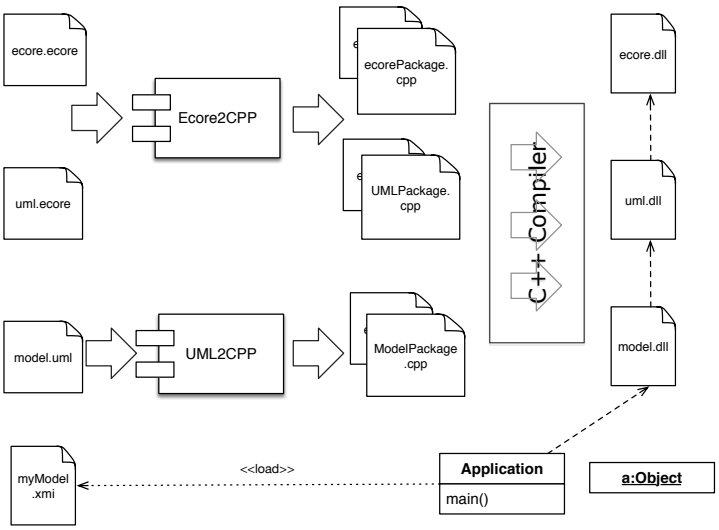

Figure 1: Structure of the generator environment.

Moreover, this generator only provides the possibility to transform Ecore models to C++ code. The transformation of the more expressive UML (unified modeling language) (OMG, 2015b) is not considered.

A native generator for C++ is necessary to use all capabilities of UML in describing the structure and behavior of a system, and to use the extension mechanism of UML by defining profiles with custom stereotypes. Such a C++ generator has to create the classes, meta-model packages, and factories to realize modeling, transformation, validation, and comparison of UML models.

This paper presents an EMF-like UML Generator for C++ (UML4CPP) aiming at a workflow and structure similar to the EMF for C++ targets. Like the EMF4CPP generator we use the Eclipse Modeling Project to formally describe the transformation with Acceleo (Eclipse, 2014). Acceleo is a generator tool which uses OMG's model-to-text transformation language (OMG, 2008). Section 2 presents the workflow and the structure of the UML4CPP Generator. Selected challenges and implementation details are presented in Section 3. The results of a comparison between the Java and C++ implementation is given and discussed in Section 4. Finally, ongoing and further work is summarized in the conclusion.

2 WORKFLOW

This section sketches the different steps which are necessary to achieve a UML-compliant C++ representation of a model. It is not enough to just convert the classes to C++ code. For a full-featured implementation, the upper levels of the model hierarchy are needed as well.

Figure 1 depicts the structure and dependencies of the elements which are necessary.

It shows the four-layered meta-model architecture of the UML. Starting with the topmost level, the meta meta-model is in our case the Ecore model. The Ecore model is transformed to C++ source code with the help of an Ecore-to-C++ generator written in Acceleo.

Ecore's expressiveness is much lower than UML. Therefore, it only can describe the structure of a software system but not the behavior. The generator needs additional information concerning the semantics of the model to generate proper source code. Therefore, both the ecore.ecore and uml.ecore models are enhanced with annotations. This is a usual method for extending Ecore models with additional informations.

Annotations are used to specify the following aspects:

- Method implementation
- Visibility of methods
- Getter/setter renaming
- Union
- Subsets
- Ignore
- C++ includes

By using the same meta meta-model and meta-model as the Eclipse modeling project, the EMP tools can be used to model, transform, and validate the input for our generator.

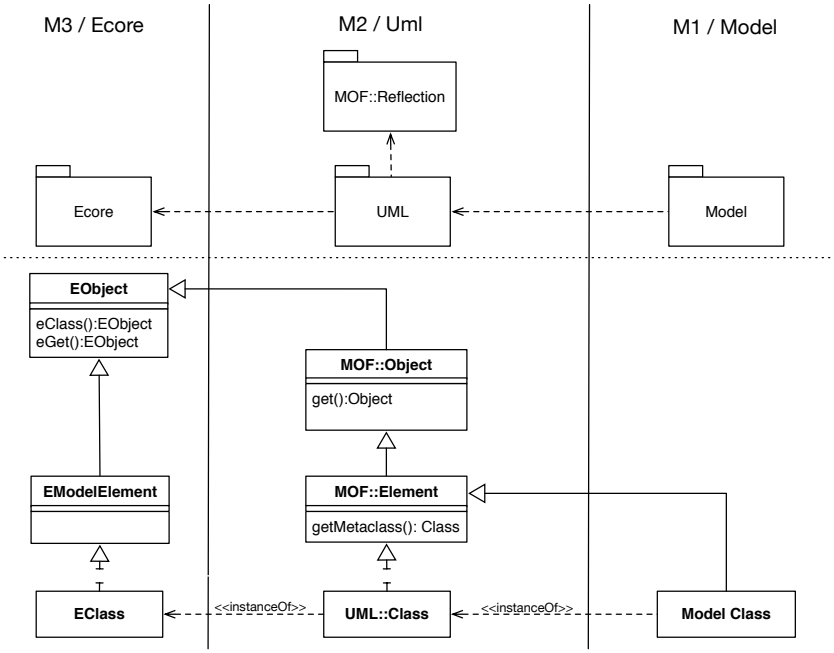

Figure 2: Excerpt of the OMG layer model, with meta meta-model, meta-model, and model layer.

The Ecore-to-C++ generator creates C++ classes, the model factory, as well as the model package. This code is compiled to a library and can be used by the lower model levels (see Figure 1).

To create source code out of UML models, a second generator is needed. It is loosely based on the Acceleo generator for Ecore, but does not need additional information except for the includes.

Until now, the supported features of our UML4CPP generator are:

- Class creation + Attributes
- Operation implementation by using UML *OpaqueBehavior*
- Reflection Package + Factory
- Profiles with Stereotypes
- Constraints
- Activities and Actions

With these generated libraries an application now can load an xmi representation of a model. This is used to create M0 instances of the model elements, which can be queried for properties of the upper layers depicted in Figure 2.

The UML meta-model in the M2 layer has a special property: Because it was described with the Ecore meta meta-model on the one hand, and provides MOF-based reflection for the lower levels on the other hand, the UML library has elements for the reflection of both meta meta-models. Because each model element includes the methods of the reflection from the upper layer, they can access the description of their own type. In the M1 layer of Figure 2 the *Model Class* can access the M2 layer by using the method *getMetaclass()*. In the M2 layer *Class* has to use the ecore reflection by using the method *eClass()* to access the M3 layer and gain the type information of itself. The mechanism of reflection is an essential aspect when using generated code of meta models in generic software. By using reflection, the program do not need to know the structure of an object. Instead the parts can be queried via the meta layer during runtime.

Figure 3 depicts the reflection as it is defined in OMGs MOF standard (OMG, 2015a). It consists of tree classes, *Object*, *Element* and *Factory*. The Object is the base Class for every Element in the Metamodel. It provides the methods to access features of an object like properties or methods. The Element extends the Element metaclass of the *UML*. It provides the capability to access the upper meta level of an meta model. That means, when we create an instance of a class we can access the description of the Class via the *getmetaclass* method of the instance. Therefore we can query the description to gain access, discover and manipulate properties and methods of the instance. The *Factory* is the creator class for the metamodel. By merging the *UML* package to the MOF reflection package, the new elements are integrated into the resulting package and the exiting once

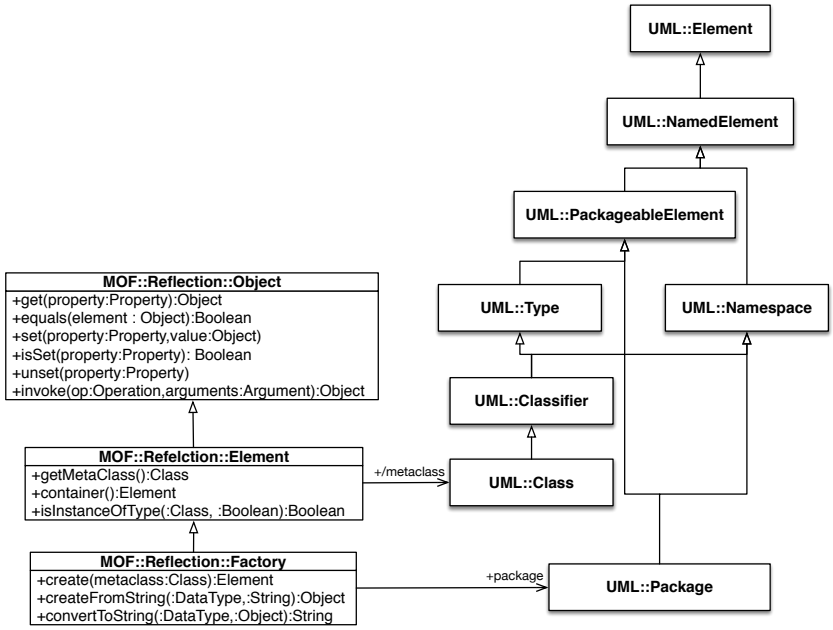

Figure 3: Overview of the elements which provides the reflection capability of the MOF.

are combined with the new elements. In the end excerpt of the *UML* is extend by the aspect of reflection.

Why is it necessary to have an additional generator, when it is also possible to use a transformation from *UML* to ecore and then use the emf4cpp generator, like eclipse does in their workflow?

Our goal is to describe the structure and the behavior of a software system with a domain specific language. When converting a model from *UML* to *ecore* or *MOF* the model is mapped to the elements of the meta meta model. Because there are no equivalent elements in the meta meta models for each element of the *UML* the amount of elements is cut. Our generator does not map the elmenets to the meta meta model. Instead it uses the whole *UML* meta model for the generation. The output of the C++ model representation is equal to an output of the emf4cpp generator when converting the *UML* model to *ecore* before the generation. The difference lies in the meta model package, which in addition to the ecore model, also has the description of behaviors, constraints and stereotypes.

3 CHALLENGES

The UML meta-model uses multiple inheritance extensively. To reduce the amount of duplicated code, which accrues when implementing multiple inheritance with single inheritance, it is advantageous to use a programming language which supports the use of multiple inheritance such as C++ instead of Java. Figure 4 depicts an excerpt of our inheritance hierarchy.

The Figure shows the strict separation between interface definition and specific implementation of the meta-model elements. This has the main advantage to provide a public API to be used easily in other projects. It provides an encapsulation and increases the loose coupling. Otherwise, a compilation of the whole meta-model would be necessary, which may take significant time.

The separation of interfaces and implementation leads to a double diamond structure of the inheritance hierarchy. The virtual inheritance has to be used in C++ for this reason. This requires the use of dynamic casts to go up and down in the hierarchy. Other casts like static or reinterpret casts are illegal or lead to an invalid behavior caused by different pointer addresses. As a result, the computation time for necessary dynamic casts is much higher in a multilevel inheritance hierarchy (Didriksen, 2010).

The C++ 11 standard provides many new syntax features to simplify programming. One of the main features is the lambda expression. These expressions are type-safe nameless functors used to simplify the implementation of Factories, Feature getters, and setters (see Listing 1). Moreover, they allow to use a similar syntax as the OMG specification.

Figure 5 presents an example UML model, representing an overview model of a so-called simulation-based application for planning and simulating wire-

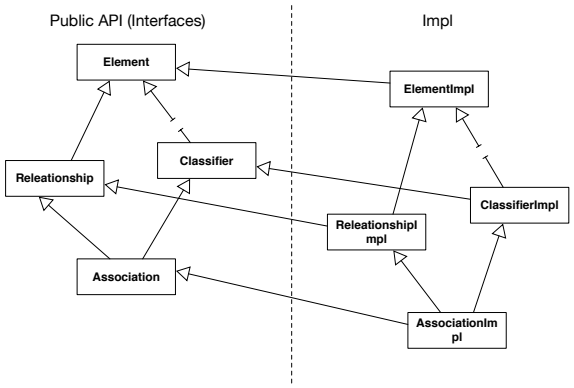

Figure 4: Double diamond model structure in the inheritance hierarchy.

less sensor networks in aircrafts ((Jäger et al., 2014), (Jäger et al., 2015)) as an example.

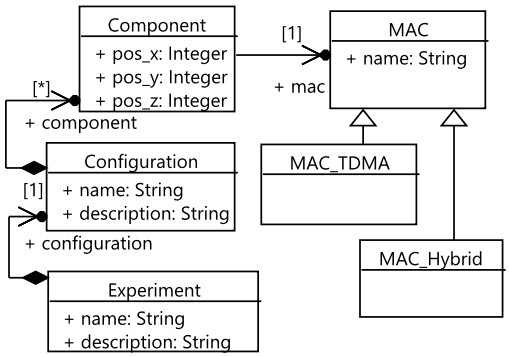

Figure 5: Excerpt of the model of a simulation based application for planning and simulating wireless sensor networks in aircrafts.

The corresponding factory source code, which was generated by the UML4CPP generator, is shown in Listing 1.

A similar method was used to implement the get(), set(), and invoke() functions for the class *Object* of the *MOF::Reflection* package. Because C++ does not have a base type like the Object in Java, these operations need a special return or parameter type for allowing both values or pointers as arguments. For this task the boost library provides an *any* data type, which can store any kind of type. The main disadvantage is the obvious lack of built-in polymorphism for this type.

Due to the lack of polymorphic containers, a special list container is needed, which allows the cast of its contents. Until now it is necessary to know the exact datatype. This issue will be changed in further progress of the development by introducing a polymorphic container.

Memory management is a considerable concern for large models. To avoid memory leaks, *std::shared_ptr* are a good solution because they automatically free memory when nobody is using an object any more. A special case is the return of a self reference, which is not applicable in every situation where self references are needed and thus could cause failures. Moreover, the overhead of reference counting leads to an increasing computation time compared to raw pointers. Thus the UML model provides the needed information for the responsibility of the instantiated elements, making it is easy to generate a safe memory management (González et al., 2010). The UML4CPP generator thus only creates source code with raw pointer. By considering the composite feature of a property, the generator creates a correct memory management as it is modeled.

4 BENCHMARK

This section presents results of a comparison between Java and C++ meta-models generated by EMF and UML4CPP. Several important properties influencing the performance of practical application are compared, including

- Creation time of model elements
- Time effort to move model elements
- Comparison of model elements
- Amount of memory used

4.1 General Settings

A standard windows PC (Intel Core i5-2520M, 2.50GHz, 8 GB RAM) is used to run the benchmarks. The Java benchmark is running on a separate Java virtual machine. It is configured in a way that only one kernel is used, because current C++ implementation dont use multi-kernel functionality. The added start-up time and size of the virtual machine itself is not considered in the following benchmarks.

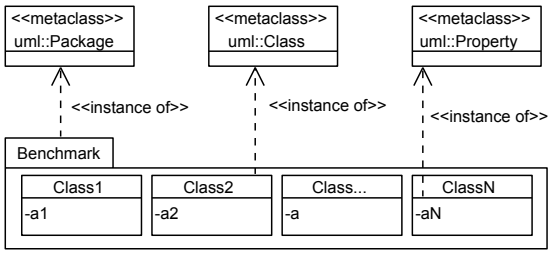

Figure 6: Simple model for the benchmarks.

```
ModelFactoryImpl::ModelFactoryImpl()
{
  m_creatorMap.insert("model::MAC_Hybrid",[this](){return this->createMAC_Hybrid();});
  m_creatorMap.insert("model::Configuration",[this](){return this->createConfiguration();});
  m_creatorMap.insert("model::Component",[this](){return this->createComponent();});
  m_creatorMap.insert("model::Experiment",[this](){return this->createExperiment();});
  m_creatorMap.insert("model::MAC_TDMA",[this](){return this->createMAC_TDMA();});
  m_creatorMap.insert("model::MAC",[this](){return this->createMAC();});
}
```

Listing 1: Generated factory code with lambda expressions for the model in Figure 5.

The benchmarks use instances of a UML test class which includes one attribute (see Figure 6). The name of the class and attribute is set. Different benchmarks use different amounts of instances of this test class.

4.2 Creation Benchmark

The first benchmark compares the creation time and the memory use of Java and C++ meta-model implementations. Therefore, instances of the described test class are created in a loop, varying between 10.000 and 100.000 instances. To analyze the influence of the Java garbage collection, the results of Java standard behavior and the results of a forced garbage collection are compared additionally. The calculation is averaged over ten runs, to average over the huge variation of computation times for the Java meta-model implementation. Figure 7 depicts the results of the first benchmark.

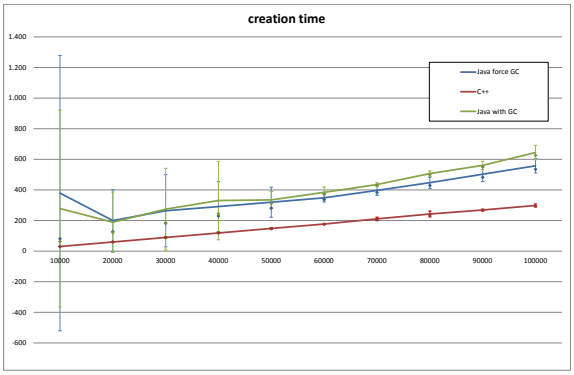

Figure 7: The graph depicts the results of the creation benchmark for C++, Java, and Java with forced garbage collection. The error bars displays standard deviation values and the markers (diamond-shape) represents median values.

The Java implementation (green and blue lines in Figure 7) need more creation time than the C++ implementation (red line). Moreover, the standard deviation of the creation time for the Java implementation is significantly greater than for the C++ implementation. Especially the first result of the first run is significant greater than the other creation times, which could be seen in the huge standard deviation of the first creation time. This is caused by the first memory allocation of the Java virtual machine and the fact that it reuses the already allocated, but not needed memory for the next runs. The influence of a forced garbage collection (blue line) is less significant than expected. The creation time of the C++ implementation is nearly linear and has a small standard deviation.

The memory clean-up of the Java implementation is faster than the C++ implementation for a huge amount of elements, because C++ deallocates the memory completely in contrast to Java.

4.3 Model Change and Undo Benchmark

To compare the performance of Java and C++ meta-model implementation regarding model changes, the movement of model elements are analyzed. Therefore, 100.000 instances of the described test class are created. All attributes are moved to one class. These operations are repeated ten times again to calculate an average effort for the movement and undo operation.

The time effort for traversing the model are analyzed by using a model comparison as well. For this purpose, 1.000 instances of the test class are created in two packages. All instances of the test class of one package are searched in the other package. This operation is also repeated ten times to calculate the mean time effort.

Table 1 shows the results of these benchmarks. It becomes clear that the Java implementation is much faster than the C++ implementation. The movement operation is almost four times faster and the model comparison is up to 100 times faster than the actual C++ implementation. The standard deviation of the Java implementation, however, is relatively high.

This unexpected result can be explained by the above-mentioned need of dynamic casts by using this multilevel inheritance hierarchy. The profiler

confirmed the already known behavior of dynamic casts (Didriksen, 2010). With a higher inheritance hierarchy, more time is needed for dynamic cast operations. Possible changes of the C++ implementation to improve this situation are discussed in Section 5.

4.4 Memory Footprint

The hard disk memory use of the implementations are compared in Table 1. The C++ implementation needs nearly three times more hard disk memory than the Java implementation. The reason for this behavior is probably the linking with several libraries including qt and boost. On the other hand, the value of the Java implementation does not contain the memory which is needed by the Java virtual machine.

The RAM use is measured by executing the benchmark of Section 4.2. The maximum required memory is captured. As a result, the Java implementation requires approximately twice as much RAM as the C++ implementation.

5 DISCUSSION

In summary, the comparison of the generated Java and C++ implementation of the test class by using several benchmarks has shown divergent results: On the one hand, the C++ implementation requires less memory than the Java implementation and is slightly faster in creating new instances. On the other hand, the current C++ implementation needs up to 100 times more time by using the model. This is caused by the need of dynamic casts in the multilevel inheritance hierarchy. To solve this issue, the dynamic cast should be avoided by using interface realizations instead of multiple inheritance. The implementation of base classes has to be generated several times into the implementation of the derived classes then. This could be realized by changing the UML4CPP generator accordingly. This demonstrates one of the advantages of a model-based generative software development.

After additional optimization steps to reduce time and memory consumption of the generated implementation, model-based applications for systems with reduced computing and storage capacities such as embedded systems could be realized.

6 APPLICATION

As already mentioned in Section 1, the generated C++ *UML* meta-model could be used in several use cases. It is the base for the development of an C++ based meta modeling tool family.

By using our approach it is possible to define domain specific languages for legacy software systems which do not have a explicit meta model right now. After adapting the legacy software to generated meta model it is now easy to optimize and extend the language. Moreover, because the meta models are based on a standard meta metamodel it is easy to transform it to another metamodel. In addition it is possibile to read and write models using XMI which is the standard description of UML.

Theoretically, it is possible to build a modeling toolchain based on C++ similar to the Eclipse Modeling Project which has components for transformation, generation and visualization.

As an example we use the generation of a *UML* model to define visualization properties for attributes of a software system shown in Figure 5 (Mambally Das, 2015). The stereotypes for visualization properties are defined via a profile diagram and are generated with the generator as well as the metamodel where the stereotypes are applied to the classes. During runtime a component analyse the classes and visualize the attribute values in a proper way. More over, it is possible to define constraints by using the standard Object Constraint Language (OCL). By using the generated information of the constraints in the meta model package, it is possible to check the values of the attributes against the defined constraints during runtime.

Another use case of C++ *UML* meta model is the model based definiton and execution of behavior, for instance by using activity diagrams (Chandrasekaran, 2015). The OMG specify a subset of the *UML* for executing activity diagrams which is called *fUML* (OMG, 2013). It also contains a model based description of the execution engine. Based on these model the generator can create the execution engine (Ramyashree, 2015). During runtime the execution engine use the meta model package to retrive the behavior elements and runs the activities.

7 CONCLUSIONS

This paper presented an EMF-like UML Generator for C++ (UML4CPP). The workflow, structure and some implementation challenges of the UML4CPP generator are presented. Examples have proven the practical usability of the resulting C++ implementations. The comparison of generated Java and C++ implementations of models by using several benchmarks w.r.t. resource use has shown diverging results.

Table 1: Results of benchmarks; * - without Java virtual machine (152 MB)

Benchmark	Java		C++	
	Mean	Std	Mean	Std
Move	375.2 ms	130.2	1405.7 ms	107.2
Compare	64.2 ms	62.5	6644.3 ms	58.9
Memory HDD	5.06 MB*		14.6 MB	
Memory RAM	201.5 MB		115.3 MB	

The presented UML4CPP generator can be used to create C++ code representations of the Ecore meta meta-model, the UML meta-model, and models based on these meta-models. This provides the basis for creating C++-based execution engines, transformation engines, OCL validation tools, and other model-based applications which are subject of future developments.

Because the generator is a frist prototype, it still needs some further development. Features like the notification framework, proxies and references to other metamodels are partly implemented or still missing.

ACKNOWLEDGMENTS

The work has been supported by the Federal Ministry of Economic Affairs and Energy of Germany under grant FKZ:20K1306D.

REFERENCES

Chandrasekaran, A. (2015). Model Driven Development of Design and Optimization of UML based Simulation Processes. Master's thesis, Technische Universität Ilmenau.

Didriksen, T. (2010). C++ dynamic_cast performance. http://tinodidriksen.com/2010/04/14/cpp-dynamic-cast-performance.

Eclipse (2014). Acceleo. http://wiki.eclipse.org/Acceleo.

EMF4Cpp (2015). What is EMF4CPP? https://code.google.com/p/emf4cpp/.

González, A. S., Ruiz, D. S., and Perez, G. M. (2010). EMF4CPP: a C++ Ecore Implementation. In *DSDM 2010 - Desarrollo de Software Dirigido por Modelos, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2010)*, Valencia, Spain.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition.

Jäger, S., Jungebloud, T., Maschotta, R., and Zimmermann, A. (2014). Model-based QoS evaluation and validation for embedded wireless sensor networks. *Systems Journal, IEEE*, PP(99):1–12. forthcoming.

Jäger, S., Maschotta, R., Jungebloud, T., Wichmann, A., and Zimmermann, A. (2015). Model driven development of simulation-based system design tools. submitted for publication.

Jungebloud, T., Jager, S., Maschotta, R., and Zimmermann, A. (2013). MOF Compliant Fundamentals for Multi-Domain System Modeling and Simulation. In *Systems Conference (SysCon), 2013 IEEE International*, pages 191–194. IEEE.

Mambally Das, R. (2015). OCL based Constraint Validator for Model-Drivern Development of Simulation based Application. Master's thesis, Technische Universität Ilmenau.

MISRA (2008). MISRA C++. Technical report. http://www.misra.org.uk/.

OMG (2008). MOF Model to Text Transformation Language 1.0. Technical report, Object Management Group.

OMG (2012). Object Constraint Language (OCL). Version 2.3.1. Technical report, Object Management Group.

OMG (2013). Semantics of a Foundational Subset for Executable UML Models (FUML) 1.1.

OMG (2014). Model driven architecture (mda) mda guide rev. 2.0. Technical report, Object Management Group.

OMG (2015a). Meta object facility (mof) 2.5 core specification. Technical report, Object Management Group.

OMG (2015b). Unified Modeling Language TM (OMG UML), Version 2.5. Technical report, Object Management Group.

Ramyashree (2015). Model Driven Development of fUML based Execution Engine. Master's thesis, Technische Universität Ilmenau.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

TIOBE Software BV (2015). Tiobe index. http://www.tiobe.com.