FROM UML TO ANSI-C
An Eclipse-Based Code Generation Framework

Mathias Funk
Department of Electrical Engineering, Eindhoven University of Technology
m.funk@tue.nl

Alexander Nyßen, Horst Lichter
Research Group Software Construction, RWTH Aachen University
{any, lichter}@swc.rwth-aachen.de

Keywords: Embedded & Real-Time Systems, ANSI-C, UML, Code Generation

Abstract: Model-driven engineering has recently gained broad acceptance in the field of embedded and real-time software systems. While larger embedded and real-time systems, developed e.g. in aerospace, telecommunication, or automotive industry, are quite well supported by model-driven engineering approaches based on the UML, small embedded and real-time systems, as they can for example be found in the industrial automation industry, are still handled a bit novercal. A major reason for this is that the code generation facilities, being offered by most of the UML modeling tools on the market, do indeed support C/C++ code generation in all its particulars, but neglect the generation of plain ANSI-C code. However, this would be needed for small embedded and real-time systems, which have special characteristics in terms of hard time and space constraints.
Therefore we developed a framework, which allows to generate ANSI conformant C code from UML models. It is built on top of Eclipse technology, so that it can be integrated easily with available UML modeling tools. Because flexibility and customizability are important requirements, the generation process consists of a model-to-model transformation between the UML source model and an intermediate ANSI-C model, as well as a final model-to-text generation from the intermediate ANSI-C model into C code files. This approach has several advantages compared to a direct code generation strategy.

1 INTRODUCTION & MOTIVATION

Model-Driven Engineering (MDE) has become very popular in recent times, particularly in the domain of embedded software systems. Although model-driven approaches like e.g. ROOM (Selic et al., 1994), ROPES (Douglass, 1999), or COMET (Gomaa, 2000) have been known to the embedded application domain for several years, they have never reached the very break through. According to our observations, it was not before the release of the current Unified Modeling Language (UML) standard (OMG, 2007b) - as well as the inception of the related SysML standard (OMG, 2007a) - that this trend has significantly gained impact.

One major reason for this is in our opinion today's availability of a broad range of mature, standard-conformant UML and SysML modeling tools, as adequate tool support is the precedence for a successful application of MDE, because only thereby its full benefits can be unleashed.

However, while modeling from the early requirements engineering up to the late detailed design can be quite well supported by those state-of-the-art UML and SysML tools, we noticed that support for generating code from the resulting design models is often not yet satisfying.

The main reason for this is that flexible customization of the code generation process is often not possible at all, or only with great effort, as nearly all tool vendors have their own proprietary code generation API. Customization potentials are often limited to what the vendor anticipated beforehand, which is not always what the user expects or desires.

Another reason, closely related to the first one which emerges in the domain of small embedded and real-time software systems, is that due to the hard space and timing constraints those systems have to face, code generation of mixed C/C++ code, as it is

solely offered by nearly all state-of-the-art UML and SysML modeling tools, most often is not adequate. Instead generation of ANSI conformant C code would be needed for those systems, as this still is the state-of-the-practice implementation language (van Solingen, 2004) for which approved compilers are available.

As we aim at providing mature model-driven engineering for the domain of small embedded and real-time software systems (Nyßen and Lichter, 2007), we introduced a code generation framework (Funk, 2006) (Kevinc, 2007) to support the flexible and customizable generation of ANSI-C code from UML models.

2 RELATED WORK

Many commercial MDE tools targeting the domain of embedded and real-time systems exist on the market (an inventory is provided by (Graaf et al., 2002), a quite extensive comparison of case tools targeting the embedded and real-time domain can be found in (Schätz et al., 2003)). Quite a few of those tools offer UML modeling capabilities, as the UML has become the predominant modeling language for MDE. Nearly all of those tools supporting the UML do as well offer direct code generation capabilities from UML models. However, as already mentioned, the offered generation capabilities are mostly inflexible and hard to customize. Moreover, the generation of plain ANSI conformant C code is often not supported. Instead, most of the tools deliver proprietary runtime libraries and concentrate on the generation of C++ code which is not applicable to the domain of small embedded and real-time systems.

While commercially available tools thus only marginally cover the regarded domain of small embedded and real-time systems, academic approaches do not seem to consider the domain at all. That is, there are almost no approaches investigating the generation of ANSI conformant C code from UML models. The reason for this - as we think - is that academic state-of-the-art and industrial state-of-the-practice are separated by a time gap of around 10-15 years, so that real-world problems being relevant to the regarded domain have simply gotten out of academic scope. That is, ANSI conformant C code generation from UML models has been inside academic scope in the early days of the UML (e.g. the generation of C code from state machines or from simple class diagrams), but after that not much research effort has been spent to those questions. For instance, the generation of ANSI-C code from UML composite structure or component diagrams, being introduced to the UML in its recent version to especially address embedded and real-time systems, has not been investigated yet.

The code generation framework we propose is intended to fill the described gap. It allows for rapid development of custom ANSI-C code generators as they are required by the regarded domain of small embedded and real-time systems. We see this as a prerequisite for the successful application of MDE technology.

3 REQUIREMENTS

Besides the fundamental functional requirement to support the generation of ANSI-C code from UML models, the most important requirement for our code generation framework is - as already mentioned - a non-functional, namely to be flexible and adaptable. That is, the code generation process has to be such that different transformation strategies, different application domains, as well as different target platforms are supported. Furthermore, as the code generation process may change over time, maintainance has to be as straight-forward as possible.

In consequence, we demand that the code generation framework should have a very modular architecture, separating the logical transformation process (i.e. how UML model elements are mapped to C language constructs) from the rather technical process of code generation. This way, changes to the logical transformation - which we consider to be the one that most likely changes - can be done without affecting the rather technical process of generating the C language unit files which is considered to be rather stable.

Eclipse has established itself as a de facto industry standard, and the related tools and technology projects offer a mature C/C++ development environment (Eclipse CDT tools project (The Eclipse Foundation, 2000a)) as well as underlying technology for model-driven engineering based on UML (Eclipse MDT (The Eclipse Foundation, 2000d) and GMT (The Eclipse Foundation, 2000b) technology projects). This led us to the decision to realize our code generation framework based on that technology as well. This way, it can be easily integrated with several other UML and SysML modeling tools, as various considerable tool manufacturers as IBM Rational, Borland, Mentor Graphics, Gentleware, No Magic, Embedded Plus (SysML Toolkit for Rational Software Development Platform), and others build their tools on top of Eclipse technologies.

4 CONCEPT AND DESIGN

As already indicated, we decided to do the transformation of UML models into ANSI-C code in two steps, namely a logical transformation step between the UML model and an intermediate ANSI-C model, which represents the C language elements as defined by the respective language grammar (Kernighan and Ritchie, 1988), as well as a subsequent generation step from the intermediate ANSI-C model to C code.

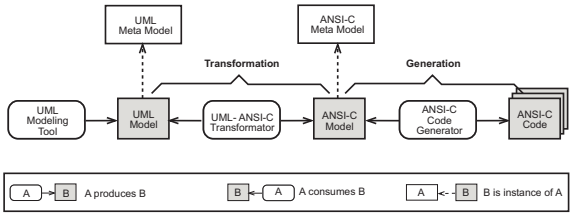

Figure 1: Code Generation Strategy

In the sense of model-driven engineering, the code generation process itself therefore consists of a model-to-model transformation between the UML source model and an intermediate ANSI-C model, as well as a final model-to-text generation from the intermediate ANSI-C model into C code files, as demonstrated in Figure 1.

This approach, using an intermediate ANSI-C model, has several advantages compared to a direct code generation strategy.

First, as the logical mapping between the UML model elements and their ANSI-C representations can be dealt with isolated from the technical domain of generating the source code files, easy adoption and customization of the transformation logic can be achieved without having to go into the technical details of the generation.

Second, as the model-to-text generation process can on the other hand be regarded as rather stable and does not have to be changed in the normal case, but can be reused as-is, this leads to increased robustness and stability of any code generator being built on top of the framework. Furthermore, increased maintainability and testability can be expected because the separation allows for maintainance and testing of both parts individually. This is in particular the case for the logical transformation process, which is realized in the technical domain of model-to-model transformation, so that the transformation strategy itself can be easily tested by checking the source and target models affected by the transformation.

Further, it can as well be ensured that the finally generated source code is ANSI conformant, as this is already ensured by the syntactical correctness and well-formedness of the intermediate ANSI-C meta model.

Such a solution offers the advantage of an overall reduced complexity. One can imagine that a combination of both steps, bringing the two problems together, would be much harder and error-prone to solve, let alone the reduced quality in terms of testability and maintainability. In this sense, the decision to split the overall process into a transformation and a generation part already simplifies the development, adaption and extension of the framework.

4.1 ANSI-C meta model

Having motivated the need for an intermediate ANSI-C model, the conception of a respective ANSI-C meta model was an essential part in the realization of our framework. Indeed, its realization is quite straightforward, as it has to represent the whole C language grammar (Kernighan and Ritchie, 1988). According to this, a C translation unit (which corresponds to a .h or .c file) contains three main language items, namely preprocessor constructs, language constructs (i.e. abstract syntax tree elements), and annotations (comments) that may be attached to both types of constructs.

The corresponding part of the meta model is displayed in Figure 2. It shows the integration of preprocessor constructs and abstract syntax tree elements, as well as annotations, which can appear admixed in the source code but which are handled each during different stages of the compilation process. They will therefore not interfere during compilation, but have to coexist in the model. As it has to be possible to add preprocessor elements before, after, and inside (structured) AST elements (e.g. inside a `ForStatement`, before the first nested `Statement`), and as it further possible to add annotations before, after, and inside (structured) preprocessor statements as well as AST elements, respective relationships between the meta model elements were designed (in case of a structure `ASTElement` or `PreprocessorElement`, `innerLeadingPosition` and `innerTrainlingPosition` refer to the positions directly before the first nested element as well as directly after the last nested one; in case of unstructured `ASTElements` and `PreprocessorElements` those associations are undefined).

Down from the `PrepocessorElement` and `ASTElement` meta model elements, the construction of the ANSI-C meta model is a pretty simple translation of the respective C language grammar elements, so we do not present this in very detail here. `Annotations` are more interesting as they

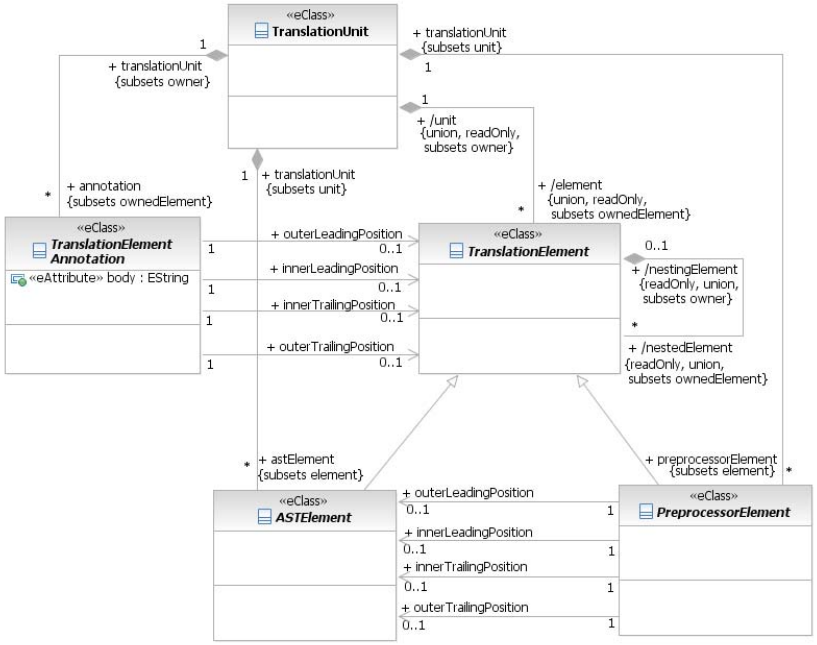

Figure 2: Core of the ANSI-C Meta Model

may appear in the form of `Comments` as well as `ProtectedRegionAnnotations` (cf. Figure 3).

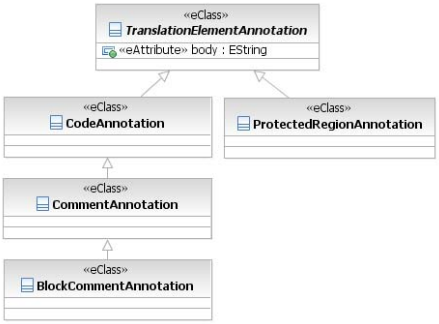

Figure 3: Annotations Hierarchy

`Comments` have to be represented in the ANSI-C meta model to offer the possibility to transport knowledge that is expressed "non-structurally" from the UML model to the ANSI-C model. `ProtectedRegionAnnotations`, have special semantics regarding the code generation process, namely to mark sections inside the source code where custom code can be inserted, being preserved from manipulation by subsequent re-generation runs. We stress this point, as this is a prerequisite for the practical usability of a code generation framework, as thereby users can alter, specialize or remove generated code and tailor the constructs to their needs.

The C language grammar naturally just defines the structure of elements embedded into a single translation unit (.h or .c file). As a complete model-to-model transformation will have to deal with a potentially high number of translation units, additional structuring means are needed. That is why the container hierarchy as displayed in Figure 4 was introduced. Containers allow to structurally organize translation units. They will be transferred into file system folders during the final generation of C code, and thus allow to specify which translation units are grouped together into folders.

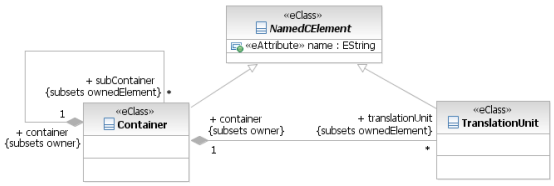

Figure 4: Container Hierarchy

As already indicated in the introduction, we decided to use Eclipse technology for the technical implementation of the ANSI-C meta model. Be-

sides Eclipse has become generally accepted in practice, the fact that an Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2000c) based implementation of the UML meta model is provided by the Eclipse Model Development Tools (MDT) (The Eclipse Foundation, 2000d) project, was a significant driver for this decision. This way, the model-to-model transformation can be easily realized whithout leaving the technical domain of EMF.

4.2 Model-to-model transformation

As already elaborated, the logical mapping from UML elements to their ANSI-C representations is realized by a model-to-model transformation between a UML input model and an ANSI-C output model. This allows us to separate the logical aspects of the code generation process from the rather technical process of generating the source code ouput files.

As both input and output models are based on EMF technology, the realization of such model-to-model transformations can be achieved quite easily with the technology offered by the Eclipse Generative Modeling Techniques (GMT) project (The Eclipse Foundation, 2000b).

We decided to use the thereby provided openArchitectureWare (OAW) framework for this purpose. It provides powerful mechanisms to operate on EMF-based models, by offering three script languages to specify model-to-model and model-to-text transformation and generation, as well as for checking models.

```
ansic::Container createPackage(uml::Package p) :
  /* create a new container */
  let d = new ansic::Container :

  /* provide the name of the newly created container */
  d.setName(p.Name()) ->

  /* process nested packages recursively; add their
     container representations to subcontainer list */
  d.subContainer.addAll(p.nestedPackage.createPackage()) ->

  /* classifiers contained in the package are transformed
     into sub containers, which contain .h and .c transl.
     units for the actually realizing the classifier */
  d.subContainer.addAll(p.getGeneratedClassifiers()
                        .createClassifier()) -> d
;
```

Figure 5: xTend Code for Transformation of UML`Package`

A model-to-model transformation can in this context be described in the form of transformation rules, being specified in the functional oAW *xTend* script language. The example in Figure 5 shows a transformation rule for a UML package (and its nested packages and classifiers).

4.3 Model-to-text generation

While the logical information about the transformation of UML model elements into their respective ANSI-C representations is encapsulated by the model-to-model transformation, the actual code generation step is a simple model-to-text generation process. It is rather straight-forward because the ANSI-C model is simply different representation of the translation unit's syntax tree structure.

As before, oAW technology is employed to realize the model-to-text generation. That is, the translation unit as well as each of its contained language elements are generated based on oAW's *xPand* language. *xPand* is a template based language, i.e. the text fragment that is generated for each model element is specified in form of a parameterizable text template. An example for an *xPand* template is shown in Figure 6. It depicts the template for a translation unit which is transferred into a file containing a top comment. Generation of contained preprocessor and AST elements is delegated to respective templates.

```
«DEFINE E FOR ansic::TranslationUnit»
   «FILE container.getFullFilePath() + "/" + name»/
     ************************************************
     * File: «name»
     * Generated by ViPER.Codegen
     ************************************************/

     «EXPAND ASTElement::E FOREACH
                  getOrphanPreprocessorStatements()»
     «EXPAND ASTElement::E FOREACH astElement»
   «ENDFILE»
«ENDDEFINE»
```

Figure 6: xPand Code for Generation of `TranslationUnit`

5 FRAMEWORK ARCHITECTURE

In order to use a code generator in daily work, it has to be integrated with other development tools (compiler, editor, etc.). We decided to realize our framework in the form of extensible Eclipse plug-ins. This does not only allow for integration into the Eclipse IDE itself, but as well into other development environments built on top of Eclipse. We decided to

split the architecture of our framework into two subsystems, the *core*, which realizes the underlying transformation and generation processes, and the *user interface*, which queries information from the user and controls the transformation and generation processes. We will describe both in the following.

5.1 Core Subsystem

The model-to-model transformation and the model-to-text generation are realized in this subsystem by respective transformation and generation units, as depicted in Figure 7.

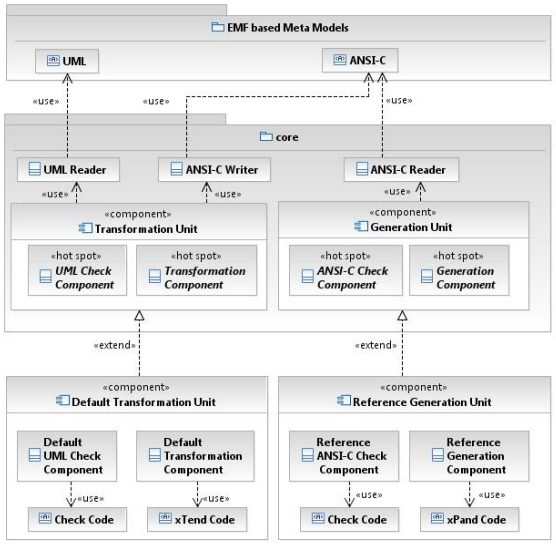

Figure 7: Core Subsystem Architecture

Reading input source models as well as writing output target models is realized with the help of model readers and writers which are specializations of generic oAW provided XMI readers and writers. *xTend* and *xPand* scripts which realize the actual transformation and generation respectively are executed by the oAW workflow engine.

To be able to customize and extend the framework constituents, both units offer extension mechanisms where customized components for model-to-model transformation and model-to-text generation can be inserted. While the model-to-model transformation is regarded to be subject to changes quite often during development (most customizations will affect this part of the over-all code generation process), the model-to-text generation part is regarded to be rather stable, as it translates already similar structures from the model representation into the textual represantation of source code. However, it might be that customization is needed there as well, e.g. because runtime libraries or platform-specific files have to be generated together with the source code. Therefore we provide an extension mechanism for this part as well.

All extensions have to offer a check component (realized by an oAW Check script) which is used to verify that the respective source model is a valid input model for the respective transformation or generation step, and a transformation respectively generation component, which bundles the *xTend* or *xPand* scripts realizing the transfomation respectively generation.

We developed so-called contributions to both extension points to demonstrate the capabilities of the framework. The reference implementation for the model-to-text generation part is regarded to be an essential part of the framework itself as it can be reused in different code generation scenarios. In contrast, the default implementation contributing the model-to-model transformation is understood as a mere demonstration example. It will most likely be replaced with a custom transformation by end users of the framework. We used it to demonstrate the capabilities of the framework by generating ANSI conformant C code from the structural information captured in a UML model.

5.2 User Interface Subsystem

The core transformation and generation units are integrated into the Eclipse with the help of wizards, which are called from respective context menus integrated into the Eclipse navigator view. This gives direct access to the desired transformation respectively generation functionality when a UML and ANSI-C model is selected as a resource in the Eclipse workbench. Figure 8 gives an overview on how this integration is achieved with the help of so called *ActionDelegates*.

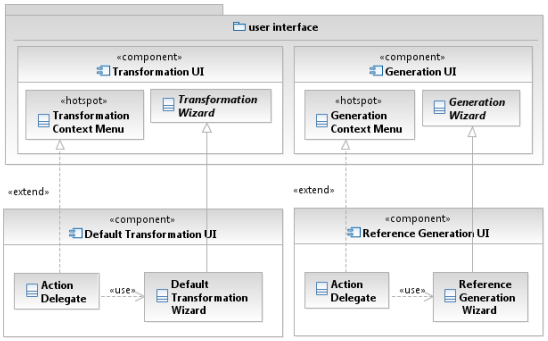

Figure 8: User Interface Subsystem Architecture

By choosing the respective context menu entry, a

wizard is started and guides the user through the transformation or generation step. In both cases, the selected input model is first validated using the provided check component. After that, further parametrization data needed for the transformation respectively generation step is collected and passed to the underlying core component. Afterwards, the execution of the respective core component is started from the wizard and the results of the transformation or generation step are displayed to the user, giving the possibility to cancel the operation.

6 EVALUATION

We evaluated our code generation framework by implementing a platform-dependent code generator. The code generator produces customized ANSI conformant C code for the Renesas M16C target platform (Renesas Technology, 2008), which is a typical representative for the domain of small embedded and real-time systems. Based on the default transformation unit provided by the framework, the M16C generator supports the generation of ANSI conformant C code from the structural information contained in a UML model, i.e. the structural information contributed to a UML model by means of UML structure diagrams.

The concept of customizability was proofed so far as the customization did indeed affect only the transformation part of our code generator and was therefore restricted to local changes only. The reference generation was not subject to change because it solely dependends on the intermediate ANSI-C model and does not contain any platform dependencies.

Experience thus showed that the separation of transformation and generation logic is a reasonable approach. In contrast to the standalone development of a comparable code generator, the overall implementation effort could be significantly reduced, as all technical details related to the generation process could be delegated to the reference generation unit. This would even hold if the transformation unit were developed from scratch, not based on the default transformation unit provided with the framework.

What can of course not be concealed is that the implementation the transformation logic entails an initial learning effort. This holds for the technical implementation, which is done in terms of developing oAW *xTend* scripts, as well as for the logical transformation, which requires knowledge about the ANSI-C model structure. This has to be taken into consideration.

Statements about the quality, performance, or maintainability of the generated code, as well as of the effort related to the construction of concrete input UML models for this specific code generator can of course not be generalized to properties of the overall code generation framework. However, it can be stated that - as experience showed - the maintainability and customizability of the generated code depends much on the appropriate usage of `ProtectedRegionAnnotations` in the design of the logical transformation process. They mark those source code locations which should not be affected by re-generation runs of the generator and thus allow the user to safely insert own code fragments. To this extend, the quality and robustness of the generated code directly depends on how well necessary additions or changes to the generated code are anticipated in the transformation logic design.

7 SUMMARY & CONCLUSION

As we stated in the introduction, we see that model-driven engineering based on the UML is getting more and more impact in the domain of small embedded systems. Due to the fact that small embedded systems are subject to hard space and timing constraints, Java code generators are not appropriate. Furthermore, C++ also has not saturated the field yet. In this sense, the C language is still the primary programming language in this domain.

We believe that the full benefits of a model-driven engineering approach can only be unleashed with the help of adequate tool support. This is the motivation for the need for a code generator, supporting the generation of ANSI conformant C code out of UML models. Our solution is a flexible and customizable framework to support the realization of custom ANSI-C code generators. As Eclipse has become the predominant integrated development environment, we decided to have our framework residing in that technical domain as well in order to achieve a seamless integration with such tools.

The framework as well as the custom code generators used for evaluation purposes, are made available as part of the ViPER platform being offered by our group. They can be obtained from the ViPER project site (RWTH Aachen University, Research Group Software Construction, 2005). We are convinced that our contribution supports the adoption of a model-driven engineering approach in the domain of small embedded systems, which is currently not the main focus of UML modeling tool vendors, but which is a domain, where model-driven engineering can improve a lot, showing all its advantages.

REFERENCES

Douglass, B. P. (1999). *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley - Object Technology Series.

Funk, M. (2006). Generating efficient C-code from UML2 composite structure diagrams. Diploma Thesis (German), RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Mathias_Funk_Thesis_Report.pdf.

Gomaa, H. (2000). *Designing Concurrent, Distributed, And Real-Time Applications with UML*. Addison Wesley - Object Technology Series.

Graaf, B., Lormans, M., and Toetenel, H. (2002). Software Technologies for Embedded Systems: An Industry Inventory. In *PROFES '02: Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, pages 453–465, London, UK. Springer-Verlag.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language - Second Edition*. Prentice Hall - Software Series.

Kevinc, O. (2007). Generating efficient C-code from UML2 behavior diagrams. Diploma Thesis (German), RWTH Aachen University, http://www.swc.rwth-aachen.de/lufgi/teaching/theses/completed/Oezguer_Kevinc_Thesis_Report.pdf.

Nyßen, A. and Lichter, H. (2007). MeDUSA - MethoD for UML2-based Design of Embedded Software Applications. Technical Report AIB-2007-07, RWTH Aachen University.

OMG (2007a). OMG Systems Modeling Language (OMG SysML) Specification. OMG Proposed Available Specification 07-02-04. http://www.omg.org/docs/ptc/07-02-04.pdf.

OMG (2007b). UML Superstructure Specification, v2.1.2. OMG Formal Document 07-11-02. http://www.omg.org/cgi-bin/doc?formal/07-11-02.

Renesas Technology (2008). M16C Family (R32C/M32C/M16C/R8C). http://www.renesas.com/fmwk.jsp?cnt=m16c_family_landing.jsp&fp=/products/mpumcu/m16c_family, retrieved: 2008/14/05.

RWTH Aachen University, Research Group Software Construction (2005). ViPER project site. http://www.viper.sc, retrieved: 2008/13/03.

Schätz, B., Hain, T., Houdek, F., W.Prenninger, Rappl, M., Romberg, J., Slotosch, O., Strecker, M., and Wisspeintner, A. (2003). CASE-Tools for Embedded Systems. Technical Report TUM-I0309, TU München.

Selic, B., Gullekson, G., and Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. Wiley.

The Eclipse Foundation (2000a). Eclipse C/C++ Development Tooling (CDT) project site. http://www.eclipse.org/cdt, retrieved: 2008/13/03.

The Eclipse Foundation (2000b). Eclipse Generative Modeling Technologies project site. http://www.eclipse.org/gmt, retrieved: 2008/13/03.

The Eclipse Foundation (2000c). Eclipse Modeling - Eclipse Modeling Framework (EMF) project site. http://www.eclipse.org/modeling/emf, retrieved: 2008/13/03.

The Eclipse Foundation (2000d). Eclipse Modeling - Model Development Tools (MDT) project site, http://www.eclipse.org/MDT. http://www.eclipse.org/modeling/mdt, retrieved: 2008/13/03.

van Solingen, R. (2004). State of the practice in European embedded software engineering. In *MOOSE seminar*, Oulu, Finland.