A METAMODEL-BASED ASN.1 EDITOR AND COMPILER FOR THE IMPLEMENTATION OF COMMUNICATION PROTOCOLS

Thomas Kistel, Ralf Vandenhouten

Zusammenfassung

In der Software-Industrie sind viele metamodell-basierte Werkzeuge entwickelt worden, um die Erstellung von Programmiersprachen und insbesondere domänenspezifischen Sprachen (DSL) zu unterstützen. Ein Beispiel für diese Werkzeuge ist Eclipse Xtext, welches eine große Popularität im Bereich der modellgetriebenen Softwareentwicklung (MDSE) besitzt. In diesem Beitrag untersuchen wir, inwieweit Xtext und andere metamodell-basierte Ansätze zur Implementierung eines Editors und Compilers für die ASN.1 Spezifikation, welche von der ITU-T standardisiert wurde, verwendet werden können. Der metamodell-basierte Ansatz zur Implementierung der ASN.1 Spezifikation ermöglicht es, ASN.1-Dokumente softwaretechnisch wie ein Modell behandeln zu können, sodass dieses ASN.1-Modell mit anderen Softwaremodellen (z. B. Zustandsmaschinen) verknüpft werden kann. Unsere Ergebnisse zeigen, dass mit relativ geringem Aufwand eine Basisimplementierung von ASN.1 zu erreichen ist, die bereits eine gute Werkzeugunterstützung liefert. Bei einigen Details der Implementierung gerät man allerdings an die Grenze des Machbaren und diese sind daher sehr schwer zu realisieren. Dies betrifft insbesondere den Parser-Generator und das komplexe Metamodell.

Abstract

In the software industry many metamodel-based tools and approaches have been developed to support the creation of programming and especially domain specific languages (DSL). An example of these tools is Eclipse Xtext, which has gained much popularity in the Model-Driven Software Engineering (MDSE) community. In this article we investigate whether Xtext and related metamodel-based approaches can also be used to implement the ASN.1 specification that was standardized by the ITU-T. The metamodel-based approach for the implementation of the ASN.1 specification allows to treat ASN.1 documents as software models, so that these ASN.1 models can be interrelated with other models (e.g. state machines). Our results show that relatively little efforts are required to create a basic implementation of this standard with good tool support. However, some details of the implementation are quite difficult to realize because they touch the limits of feasibility. This concerns in particular the parser generator and the complex metamodel.

» I. INTRODUCTION

In software engineering small languages for different domains were gaining popularity during the last years. These small languages are also referred to as Domain Specific Languages (DSLs) and are promoted by the Model-Driven Software Engineering (MDSE) community. On the other hand the Unified Modeling Language (UML) (OMG, formal/2011-08-06, 2.4.1), (OMG, formal/2011-08-05, 2.4.1) was specified by the Object Management Group (OMG) to unify various graphical diagrams for the description of software systems. The development of UML has also led to the specification of the Meta Object Facility (MOF) (OMG, formal/2011-08-07, 2.4.1) by the OMG. The extensibility mechanism of UML, which is called UML profiles, can also be used to develop domain specific modeling languages (Selic 2007). Selic (Selic 2012) also remarked that an alternative for developing DSLs is MOF, which has, however, only a few implementations (Scheidgen 2006). The most popular implementation of MOF is the Eclipse Modeling Framework (EMF) (Steinberg et al. 2009). Many metamodel-based tools for EMF have been developed to support the creation of DSLs and the corresponding domain-specific workbench.

Eclipse Xtext (Xtext Project Website) and JetBrains MPS (Meta Programming System) (MPS (Meta Programming System) project website) are two mature metamodel-based technologies for the development of DSLs and surrounding tool support (Text editor, Parsers, Syntax highlighting, Code completion and generation, etc.). Andova (Andova et al. 2012) and Völter (Völter 2011) provide a good introduction into the detailed concepts of Xtext and MPS as well as related tools and approaches. An advantage of Xtext is that it uses EMF as the basis for its metamodels. Therefore it can be easily integrated with other EMF-based tools.

Besides the growing popularity of DSLs, especially in the context of MDSE, there already exist many DSLs in different software domains, of which the database query language SQL is one popular example. The domain of protocol engineering, which is predestined for MDSE, also takes advantage of several DSLs. The most important is the Abstract Syntax Notation One (ASN.1) (ITU, X.680, 11/2008), a DSL for the description of data types and data structures together with their corresponding encoding rules. ASN.1 was standardized by the ITU-T. In protocol engineering, ASN.1 is used to describe

the messages that are exchanged between two communicating nodes.

In this article we investigate, how a metamodel-based editor for ASN.1 can be developed with the Eclipse Xtext framework. Furthermore, we analyze of how the created ASN.1 metamodel can be used for code generation. These steps mentioned have very much in common with traditional compiler construction. The research question of this article is whether the new MDSE tools around Eclipse Xtext can be used for the development of a compiler for a relatively complex language like ASN.1. The intention for the use of a metamodel-based approach for the implementation of the ASN.1 specification is that it allows to treat ASN.1 documents as regular software models. The advantage of this approach is that these ASN.1 models then can be interrelated with other software models - e.g. state machines (Kistel 2012).

The remainder of this article is structured as follows: In section 2 we discuss related approaches for the implementation of an ASN.1 compiler, in section 3 we present the result of our preliminary solution, in section 4 we conclude this paper.

» II. RELATED WORK

The original developments of ASN.1 were already done in the 1980ies, where ASN.1 was used to describe data structures for most OSI application protocols (Bochmann et al. 2010). Therefore there already exist many ASN.1 compilers from open source or commercial vendors of which the most important ones are listed on the ASN.1 project site of the ITU (ITU ASN.1 Tools). To the very best knowledge of the authors there is currently no implementation of the ASN.1 specification with a metamodel-based approach that allows the integration with other existing MDSE tools. However, there is some related work, which addresses similar aspects compared to our implementation.

In the context of models and UML, Ek (Ek 24/11/2002) proposed an UML profile for ASN.1. UML profiles are a lightweight mechanism to refine or extend the UML language with specific notations. However, a UML profile for ASN.1 is basically a mapping to a UML class diagram, which allows modelers to specify ASN.1 descriptions with UML tools. The question is, how these text-based descriptions can be effectively created and maintained with existing UML tools or how UML profiles can be combined with declarative approaches to specify UML diagrams (Spinellis 2003, Torchiano et al. 2005). More importantly with UML profiles is that it is currently not easy to determine whether a particular specialization of a UML concept is semantically aligned with the base concept of UML, so that UML tools will treat it correctly (Selic 2012). This particularly applies to the evaluation of expressions of ASN.1 data types, which need to analyze their semantics.

Another issue, which has many influences on the compiler implementation, concerns the parser technology that is used to read an ASN.1 document and to create the syntax tree. As we will see in section 3.1 the parser technology of the Xtext framework is ANTLR (ANother Tool for Language Recognition) (ANTLR website). Different implementations of the ASN.1 specification with ANTLR can be found on the grammar list page of the project website of ANTLR (ANTLR website). All these implementations only target parts of the ASN.1 specification. Furthermore, these implementations only provide lexing and parsing capabilities for ASN.1 documents. They are not integrated into editors, metamodels or code generation tools.

An implementation of an ASN.1 compiler with the Xtext framework was already done in a master thesis (Wendlandt 2010) at Wildau University of Applied Sciences in 2010. This implementation is based on an older version of Xtext (0.7) and also uses the Xpand framework (Xpand project webiste) for code generation. The present work can be seen as a re-implementation of the work in the master thesis.

» III. SOLUTION

3.1. Parsing and validation

The corner stone of implementing a language with the Xtext framework is to define an EBNF-like grammar file. Xtext uses this grammar file to generate Parser, Lexer, EMF Ecore meta model, Editor and other Eclipse workbench functions. The underlying parser/lexer technology of Xtext is ANTLR (ANTLR website; Parr op. 2007), which creates the concrete syntax of the language. The Xtext framework translates the Xtext grammar file, into a grammar description of ANTLR. Then the lexer and parser are generated from ANTLR grammar file. ANTLR is a two phase LL(*) parser. This has many influences on the implementation of the grammar, because the LL(*) parser does not allow left recursions and the ASN.1 specification (ITU, X.680, 11/2008) in contrast is highly left recursive. Therefore all left recursions have to be “left-factorized” to remove them. At the time of writing this article we have implemented a huge part of the ASN.1 specification, which allows for parsing many ASN.1 protocol specifications. The left refactoring of the ASN.1 rules guarantees that the implementation of the Xtext grammar does not need the use of the ANTLR syntactic predicates or backtracking options. However, the current implementation does not cover all parts of the `Tags`, `Constraints` and `Value Assignments` of ASN.1. Figure 1 shows the ASN.1 Editor that was created by the Xtext framework with our grammar implementation.

```
11073.20601.asn
244 -- Observed value for compound numerics
245 --
246 NuObsValueCmp ::= SEQUENCE OF NuObsValue
247 --
248 -- SaSpec describes the sample array.
249 --
250 SaSpec ::= SEQUENCE {
251     array-size INT-U16,         -- number of samples per metric update period
252     sample-type SampleType,
253     flags SaFlags
254 }
255 --
256 -- SampleType describes one sample in the observed value array.
257 --
258 SampleType ::= SEQUENCE {
259     sample-size INT-U8, --## (SIZE(8)), e.g., 8 for 8-bit samples, 16 for 16-bit samples,
260                                 -- shall be divisible by 8
261     significant-bits INTEGER {      -- defines significant bits in one sample
262         signed-samples(255)         -- if value is 255, the samples
263     } (0..255)                      -- in Simple-Sa-Observed-Value and
264                                     -- lower-scaled-value and upper-scaled-value in
265                                     -- ScaleRangeSpec shall be interpreted as signed
266                                     -- integers in twos-complement form.
267 }
```

Fig. 1) ASN.1 editor with tool support

Besides lexing and parsing, i.e. the transformation of tokens into an abstract syntax tree (sentences), another important step in the frontend phase of a compiler is the semantic analysis of the language. In the Xtext framework this can be done by validation rules. We have implemented only some simple validation rules for the ASN.1 language (e.g. check of case sensitivity in type names). Moreover, we have integrated the ASN.1 Syntax Checker tool by OSS Nokalva Inc (http://www.oss.com/asn1/products/asn1-syntax-checker/asn1-syntax-checker.html). This allows validating the complete syntax of an ASN.1 document in the background and reporting syntax errors and warnings to the user. This solution allows for parsing and validation of almost any ASN.1 file.

3.2. ASN.1 metamodel

The Xtext framework not only generates a parser and lexer through ANTLR, it also generates an EMF metamodel, which describes the abstract syntax of the language. Xtext basically has two implementation strategies for the abstract syntax: 1) create the Xtext grammar file and generate the abstract syntax (i.e. EMF metamodel) from it; 2) create the abstract syntax (i.e. EMF metamodel) first and refer the model elements in the Xtext grammar file. In our implementation we have chosen the first strategy, because we are implementing an existing language that is specified in an ITU recommendation (ITU, X.680, 11/2008). In contrast to our strategy it could be useful to first define the abstract syntax and then the concrete syntax through the grammar file, when one is implementing a new language. Nevertheless, in both strategies, there is always a one-to-one mapping between abstract and concrete syntax in Xtext. This means that every parser rule (except terminal and data type rules) of the concrete syntax is represented in the abstract syntax. The opposite, every element of the abstract syntax is represented in the concrete syntax, is not true.

Regarding our implementation, Figure 2 shows the model hierarchy of the `BuildinTypes` of the ASN.1 specification. In our compiler implementation we have introduced a distinction of `BuildinTypes`. The model element `SimpleType` was introduced for types that do not contain other types. In contrast, `ContainerType` was introduced for types that do contain other types (see Figure 3).

The `PrefixedType` is an exception here and is a new type which is isomorphic with another type, but has additional tags or encoding instructions. `SimpleType` and `ContainerType` are represented in the abstract syntax (i.e. metamodel) but are not represented in the concrete syntax of our ASN.1 implementation.

In summary the direct mapping of the concrete syntax to the abstract syntax is not a problem for simple languages, but results in large metamodels for more complex languages. In our current implementation of the ASN.1 language, the metamodel contains 127 model elements, which is relatively complex. This issue also has some influences on the code generation, which we will describe in the next section.

3.3. Code generation

Code generation is part of the backend phase of a compiler, which usually creates machine or interpreter code. In our implementation the code generation phase creates a higher level programming language (e.g. an imperative, object oriented language like Java). In MDSE terms, code generation is also referred to as a model to text (M2T) transformation (Flores Beltran et al. 2007), because generally a M2T generator may not only create programming code, but also other artifacts like configurations, database or user-interface scripts.

In our implementation of the ASN.1 compiler we use the Xtend2 language (Xtend project website), which is now

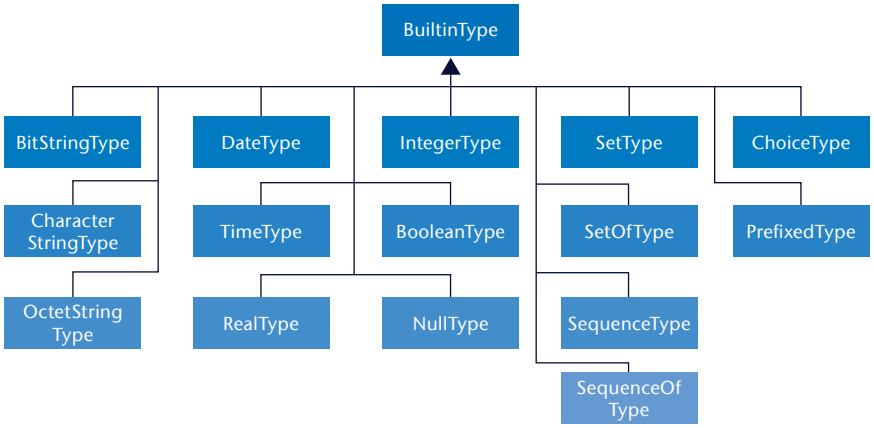

Fig. 2) Model hierarchy of ASN.1 BuildinTypes

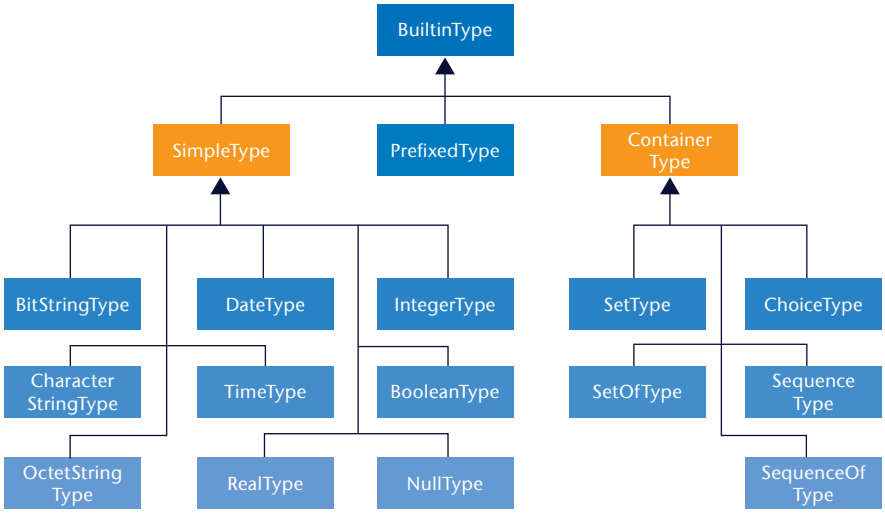

Fig. 3) Modified model hierarchy of ASN.1 BuildinType

the default generator language of the recent version (2.3) of the Xtext framework. The source of our code generation is an EMF model instance, the target language is Java7 (The Java™ Language Specification 2012). Generally different implementation subjects for the code generation have to be distinguished. These subjects are 1) the internal behavior of the generator and 2) the output configuration of the generated code.

Figure 4 shows the two different strategies for the internal behavior of the generator. Strategy a) creates an intermediate model from the source model, which can contain some optimizations etc. of the source model and then generates the target model. This way code generation is done in Xbase, which is part of Xtext and allows for reuse of expressions in different DSLs.

During the code generation, the Xbase model is inferred to a JVM types model, from which Java code is generated.

It is also possible to create a different intermediate model than the JVM types model, where necessary. In our ASN.1 compiler implementation we have done a direct conversion of the EMF model to Java code. This is possible, because the ASN.1 syntax has a strict type and naming (typereference) convention. Each ASN.1 typereference is translated into a Java class or field name and each ASN.1 type is translated into a Java class or field type. A problematic issue in the code generation is the complex traverse of the model. A simpler model would be helpful for the code generation.

The other subject (2) concerns the output configuration. In traditional compiler implementations these are options that control the runtime behavior or memory usage of the generated code. These options are also valid for code generation. The generated programming code can also be optimized for performance issues or memory management. The latter one also implies whether the generated code depends on a runtime library the generated code depends on. Traditional implementations of ASN.1 compilers for Java always use a runtime library (jar-file) which defines Java classes for the standard ASN.1 types and their encoding rules. For those ASN.1 compilers the generated type classes inherit from the ASN.1 library classes. This approach has the advantage, that every generated ASN.1 type class is explicitly defined and has a strict semantics according to the super class or its interface definition. But this approach has also some drawbacks. The most important are 1) the library also defines classes and methods that may not be required or should not be used in the target environment and 2) the inheritance reduces the flexibility of the generated classes.

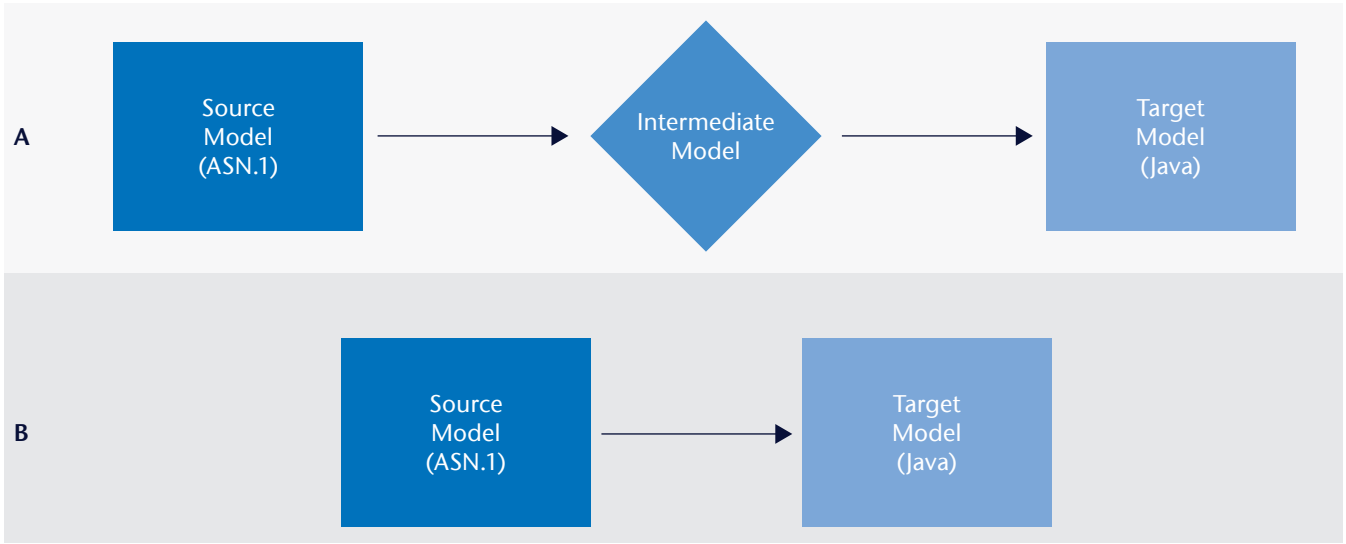

Fig. 4) Internal behavior strategies of a code generator

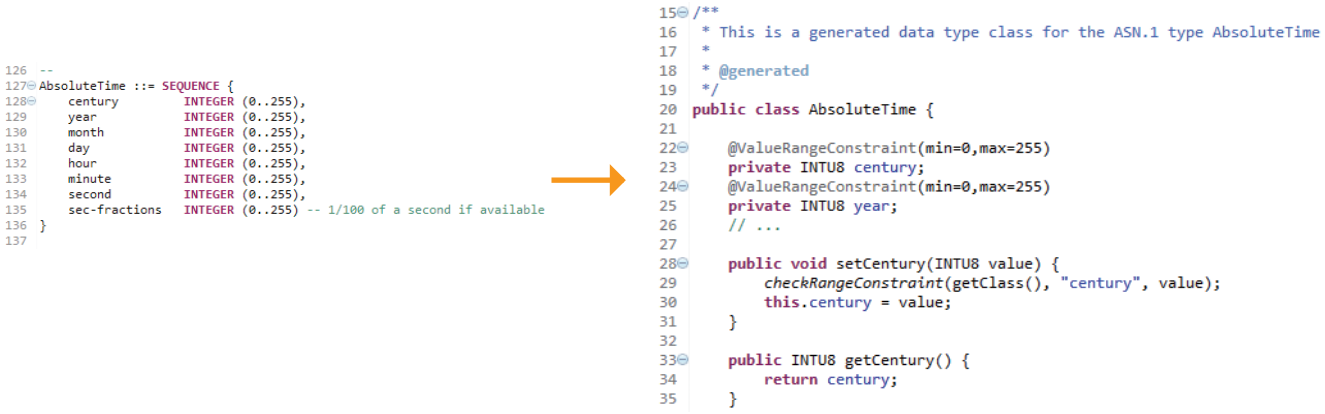

Fig. 5) Example of the code generation for an ASN.1 Sequence type

Our implementation of the ASN.1 code generator includes generation of Java classes for ASN.1 type assignments. Library classes and functions are copied into the target package as needed. This reduces the amount of the target code, because no runtime library is needed. Besides simple type assignments we also generate code for `Constraints`. Currently we generate `size constraints` and `value ranges` to Java Annotations that are attached to the corresponding Java field. These Annotations are then validated in their equivalent setter and read methods. Figure 5 shows this concept in a simple example for an ASN.1 Sequence type. The sequence type AbsoluteTime that is part of the ASN.1 protocol specification of (ISO/IEEE, 11073-20601, 2010) is generated into a Java class, the child elements of the sequence type become field members of this Java class.

the implementation provides a sophisticated IDE support, i.e. an editor with syntax highlighting, code completion, formatting and integration into the Eclipse workbench. A drawback of this approach is the strict coupling between Xtext and the LL(*) parser generator ANTLR. This requires a complex rewriting of the grammar rules to be conforming to LL grammars. An optional replacement of the parser generator, like the possible replacement of the code generator Xtend2 through Xpand, Acceleo (both Eclipse projects on http://www.eclipse.org/modeling/m2t) or others, would be helpful for this issue.
In future we plan to extend the grammar implementation of the ASN.1 specification and to extend the code generator to support different encoding rules (e.g. MDER Schrenker, Todd 2001 and BER).

» IV. CONCLUSIONS

In this article we have discussed the different aspects on implementing an ASN.1 editor and compiler. To achieve this we have used the Eclipse projects Xtext and Xtend2. With these frameworks we have implemented a huge part of the ASN.1 specification. We have proved this implementation on the ASN.1 data type description of the IEEE 11073-20601 standard (ISO/IEEE, 11073-20601, 2010), which defines an exchange protocol for personal health device communication.
Our results show that the ITU recommendation ASN.1 can be implemented with the metamodel-based tools Xtext and Xtend2. With relatively little effort

ACKNOWLEDGEMENTS

This work is funded by the German ministry for education and research (BMBF) in the ongoing project MOSES – Modellgetriebene Software-Entwicklung von vernetzten Embedded Systems (FKZ 17075B10).

BIBLIOGRAPHY

ANTLR website. Available online at http://www.antlr.org, checked on 7/09/2012.

Xpand project webiste. Available online at http://wiki.eclipse.org/Xpand, checked on 14/09/2012.

Xtend project website. Available online at http://www.eclipse.org/xtend, checked on 23/07/2012.

Andova, S., van den Brand, M., Engelen, L., Verhoeff, T. (2012): MDE Basics with a DSL Focus. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (Eds.): Formal Methods for Model-Driven Engineering. 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, vol. 7320. Berlin / Heidelberg: Springer (Lecture Notes in Computer Science, 7320), 21–57.

Bochmann, G. v., Rayner, D.; West, C. H. (2010): Some notes on the history of protocol engineering. In Comput. Netw 54, 3197 3209. Available online at http://dx.doi.org/10.1016/j.comnet.2010.05.019.

Eclipse Foundation: Xtext Project Website. Available online at http://www.eclipse.org/Xtext, checked on 15/07/2012.

Ek, A. (2002): An ASN.1 Profile. Workshop on Language Advisory Board. Geneva, 24/11/2002. Available online at http://www.itu.int/itudoc/itu-t/workshop/laboard, checked on 19/07/2012.

Flores Beltran, J. C., Holzer, B., Kamann, Th., Kloss, M., Mork, St., Niehues, Th., Thoms, K. (2007): Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis. Edited by Jens Trompeter, Georg Pietrek. Frankfurt [Main]: Entwickler.press.

ISO/IEEE 11073-20601: 11073-20601: Health informatics – Personal health device communication – Application profile – Optimized exchange protocol.

International Telecommunication Union (ITU): ITU ASN.1 Tools. Available online at http://www.itu.int/ITU-T/asn1/links/index.htm, checked on 10/09/2012.

ITU X.680, 13/11/2008: Abstract Syntax Notation One (ASN.1) – Specification of basic notation.

JetBrains: MPS (Meta Programming System) project website. Available online at http://www.jetbrains.com/mps, checked on 3/09/2012.

Kistel, Th. (2012): On the extension of message syntax languages for the development of communication protocols. In: ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems. Doctoral Symposium. MODELS. Innsbruck, 02.10.2012.

OMG formal/2011-08-07, August 2011: Meta Object Facility (MOF) Core.

OMG formal/2011-08-05, August 2011: Unified Modeling Language (UML) - Infrastructure.

OMG formal/2011-08-06, August 2011: Unified Modeling Language (UML) - Superstructure.

Oracle America Inc. (2012): The Java™ Language Specification. Java SE 7 Edition. With assistance of James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley. Available online at http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf, checked on 11/09/2012.

Parr, T. (op. 2007): The Definitive ANTLR reference guide. Building domain-specific languages. Raleigh [etc.]: The Pragmatic bookshelf.

Scheidgen, M. (2006): CMOF-model semantics and language mapping for MOF 2.0 implementations. In: Ricardo J. M. (Ed.): Fourth and Third International Workshop on Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006; 30 March 2006, Potsdam, Germany. Computer Society; Institute of Electrical and Electronics Engineers (IEEE); Workshop on Model-Based Development of Computer-Based Systems; International Workshop on Model-Based Methodologies for Pervasive and Embedded Software; MBD-MOMPES 2006. Los Alamitos, Calif: IEEE Computer Society, pp. 10.

Schrenker, R.; Todd, C. (2001): Building the Foundation for Medical Device Plug-and-Play Interoperability. In: Medical Electronics Manufacturing. Available online at http://www.medicalelectronicsdesign.com/article/building-foundation-medical-device-plug-and-play-interoperability, checked on 20/07/2012.

Selic, B. (2007): A Systematic Approach to Domain-Specific Language Design Using UML. In: ISORC ,07. 10th IEEE International Symposium on : Object and Component-Oriented Real-Time Distributed Computing, 2007, 2–9.

Selic, B. (2012): The Less Well Known UML. A Short User Guide. In Bernardo, M., Cortellessa, V., Pierantonio, A. (Eds.): Formal Methods for Model-Driven Engineering. 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, vol. 7320. Berlin / Heidelberg: Springer (Lecture Notes in Computer Science, 7320), 1–20.

Spinellis, D. (2003): On the declarative specification of models. Software, IEEE. In IEEE Software 20 (2), 96–95.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E, (2009): EMF. Eclipse Modeling Framework. 2nd ed. Upper Saddle River, NJ: Addison-Wesley.

Torchiano, M., Ricca, F., Tonella, P. (2005): A comparative study on the re-documentation of existing software: code annotations vs. drawing editors. In : International Symposium on Empirical Software Engineering 2005 (ISESE 2005). Noosa Heads, Queensland, Australia, 277–287.

Völter, M. (2011): Language and IDE Modularization, Extension and Composition with MPS. In: Lämmel, R. Saraiva, J., Visser, J (Eds.): Generative and Transformational Techniques in Software Engineering IV. International Summer School, GTTSE 2011. Pre-proceeding, 395–431.

Wendlandt, O. (2010): Entwurf und Realisierung eines Konzeptes für modellgetriebene Softwareentwicklung von Geräteschnittstellen. Master Thesis. University of Applied Sciences Wildau, Wildau. Institute of Telematics

AUTHORS

Thomas Kistel, M. Eng
Fachgebiet Telematik
Fachbereich Ingenieurwesen/Wirtschaftsingenieurwesen
TH Wildau [FH]
thomas.kistel@th-wildau.de

Prof. Dr. rer. nat. Ralf Vandenhouten
Fachgebiet Telematik
Fachbereich Ingenieurwesen/Wirtschaftsingenieurwesen
TH Wildau [FH]
ralf.vandenhouten@th-wildau.de