Verification and Validation in the Context of Domain-Specific Modelling

Janne Merilinna
VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 Espoo, Finland
janne.merilinna@vtt.fi

Juha Pärssinen
VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 Espoo, Finland
juha.parssinen@vtt.fi

ABSTRACT

The utilisation of Domain-Specific Modelling (DSM) in software development has a significant positive impact on productivity. The productivity increase is caused by the utilisation of modelling languages and generators that are especially suitable for a specific problem domain instead of those designed for solution domains. The prerequisite for this significant productivity increase is that the languages and the automation function correctly. To ensure the suitability of the languages and tools, we need to be able to use the verification and validation (V&V) techniques in the context of DSM. In this position paper we study what V&V actually stands for in this particular context and what the current means are for performing V&V. We found that although there are some means available for verification, comprehensive methods still do not exist. For validation, we believe that maintaining a bidirectional trace link between requirements, models and the generated deliverables is a promising approach to significantly facilitate the validation process.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification - *Validation*

General Terms

Design, Languages and Verification

Keywords

Model-Driven Development, Model-Based Testing and Requirements Tracing

1. INTRODUCTION

Model-Driven Development (MDD) is about treating models as first-class design entities. Modelling provides a view to a complex problem and its solutions. Models are less risky, cheaper to develop and easier to understand than the implementation of a genuine target system. Thus, the intent of MDD is to raise the abstraction level from code-centric development to model-centric development and at the same time to increase productivity by an extensive utilisation of automation, e.g. code generation, in software development.

MDD, in general, can be divided into two paradigms based on the utilised language in modelling: general-purpose modelling, in which general-purpose languages such as Unified Modeling Language (UML) are utilised, and Domain-Specific Modelling (DSM), in which Domain-Specific Modelling Languages (DSML) [1] are applied. When considering general-purpose modelling languages, there is very little evidence of productivity gains. Instead there is evidence of the opposite [2]. In the case of DSM, there is documented evidence of an increase in productivity by a factor of 5-10 [1][3][4][5][6].

Although the benefits of DSM appear to be excellent, the DSM development approach is still not very widespread. In the past, one of the great obstacles to the introduction of DSM was the lack of tooling or the tedious effort required to develop such tooling. This is not an issue with general-purpose modelling languages such as UML as there is a single standard language available, for which there is also a relatively large tool vendor community. In the case of DSM, having a large tool vendor community to support a particular DSML can be considered far-fetched as there are few reasons for standardising a language that is specific to some specialized domain. The languages and the related generators have to be developed separately for each domain, which makes it unlikely that a large tool vendor community for a language will emerge. However, nowadays there are language workbenches [7] available enabling a rapid development of languages and generators, such as the commercial MetaCase MetaEdit+ tool [8], the Microsoft DSL tool [9] and the Jviews framework [10], as well as open source tools such as Jet Brains MPS [11] running on Eclipse, for instance. An alternative approach is to develop DSMLs based on UML profiles, enhanced by a layer facilitation of the DSM definition, thus benefiting from the vast availability of these tools. In conclusion, tooling should no longer be an issue as regards language development and use. However, the quality of the developed DSMLs and the accompanied automation facilities can still be questioned.

In order to make sure that the developed DSMLs and generators are of high quality, and in this way to raise confidence in DSM, there has to be means with which to verify and validate (V&V) the DSM basic architecture, i.e. metamodels, generators and application models. A recent publication about DSM worst practices [12] provides some guidelines on how-not-to develop languages, which enable developers to avoid certain pitfalls. The iterative and incremental language development approach suggested in [1] will also facilitate the development of languages. However, if the developed language is to be deployed for more than a few developers, it would be preferable to have systematic V&V techniques in place in order to make sure that the languages and the accompanied generators function as planned. In addition, if the languages are to be utilised by many developers, having V&V techniques for the application models in place would also be favourable.

In this position paper, we take a look at what the meaning of V&V is in the context of DSM. We approach the issue by applying the IEEE Standard Computer Dictionary [13] definition of V&V and scrutinise what it stands for in the context of DSM. We identify the black spots that deserve V&V and investigate what means are currently available for V&V in this particular context. The work presented in this position paper can be seen as a continuation of the group work on V&V in the context of DSM held in the OOPSLA DSM '09 Workshop [14].

This paper is structured as follows. First, the background for discussions about V&V in the context of DSM is introduced by presenting the DSM basic architecture that needs to be verified and validated. Second, the IEEE Standard Computer Dictionary definitions of V&V are adapted into the context of DSM. Following this, state-of-the-art means for the verification and validation of various layers of DSM basic architecture are introduced. The conclusion and final remarks close this position paper.

2. DOMAIN-SPECIFIC MODELLING

In DSM, models are constructed using concepts that represent things within the application domain and not the concepts of a given programming language. The modelling language follows the domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts. The models simultaneously represent the design, implementation and documentation of the system, which can be generated directly from them. In many cases the final products can be generated automatically from these high-level specifications by means of domain-specific code generators.

The DSM solution can be seen to be constructed of a set of basic items, as depicted in Figure 1. The platform, or software framework, provides services common to all applications of the product family. Thus, in the sense of product families, the platform embeds all commonalities. The generated application then utilises the services provided by the platform. The generated applications follow the architectural and domain rules set by the product family architecture and the domain in question. The generated application is based upon input from the actual DSM solution, i.e. an application model, a metamodel and a code generator. The metamodel, i.e. the modelling language, reflects the problem domain, incorporating domain concepts and domain rules. In addition, in practice, the metamodel also includes rules from the target platform that need to be taken into account while modelling. The application model can then be modelled by applying the domain concepts and by following the rules defined in the metamodel. In an optimal case the rules are enforced by the utilised language workbench. The generator is responsible for taking the application model as the input and generating the target format from this input. The generator enforces the rules defined in the metamodel, implicitly or explicitly, and takes the rules of the target platform into account when generating the output. Thus the generated output is affected both by the rules defined in the metamodel and those in the generator.

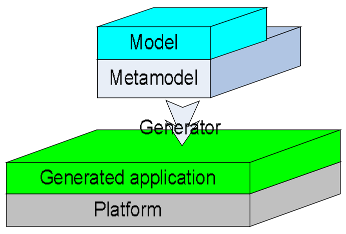

Figure 1. The DSM basic architecture.

3. VERIFICATION AND VALIDATION

The IEEE Standard Computer Dictionary [13] defines validation as:

- *'the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.'*

Model validation [13] is defined as:

- *'the process of determining the degree to which the requirements, design or implementation of a model are a realization of selected aspects of the system being modelled.'*

Verification [13] is defined as:

- *'(1) the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness.'*

Model verification [13] is defined as

- *'the process of determining the degree of similarity between the realization steps of a model; for example, between the requirements and the design or between the design and its implementation.'*

In this paper, we consider the development process as a process of transforming end-user requirements to a system fulfilling such requirements with a DSM approach. The development process includes phases for developing the modelling infrastructure, i.e. metamodels, generators and also application models, which are transformed into the target format running on a software platform.

Considering the IEEE Standard Computer Dictionary definitions, the following figures (see Figure 2 and Figure 3) can be drafted to depict what V&V means in the context of DSM. Requirements (R) means the requirements for the whole product family including the requirements for specific products and the target platform.

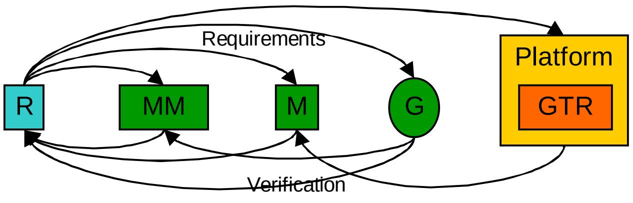

Figure 2. Verification in the context of DSM.

Here, the verification of metamodels (MM) means determining if a metamodel encapsulates 1) the problem domain (e.g. the variability space in the context of product families) including domain rules and 2) the rules of the target platform that are necessary to take into account in the metamodel. Thus, a metamodel should only allow the modelling of applications that are syntactically and semantically correct. The verification of models (M) means determining whether a model encapsulates a product under development based on the requirements set for it (e.g. a variant within the context of the product family). The verification of generators (G) means determining whether the generators produce the correct output (generated textual representation: GTR) from the input models. This involves verifying that the generators interpret the models correctly and generate syntactically- and semantically-correct GTR for those that will utilise the GTR. Verifying the GTR is about evaluating whether or not the intention of a model is realised in the outcome, i.e. in the GTR produced by a generator.

Validation is the process of evaluating the GTR, including the target platform (Platform + GTR in Figure 3), to determine whether it satisfies the specified requirements. Thus, validation can be seen as the ultimate test for the modelling infrastructure, the application models and the platform on which the application is generated. It must be noted that one application model can only be seen as one validation case for the whole modelling infrastructure.

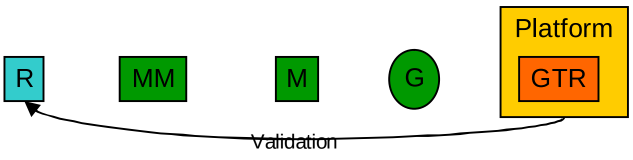

Figure 3. Validation in the context of DSM.

3.1 The Verification of Generators

The main difference between the verification of generators and the verification of traditional software components is to be found in the nature of the input data. Whereas traditional software components take integers and strings etc. as input, the generators take potentially highly complex data structures as input, i.e. models, and produce complex data structures as output. In [15], Kuster defines four aspects that need to be checked as regards model transformations: 1) The syntactic correctness of a model transformation: in order to ensure that the outcome of the transformation is syntactically correct. 2) The termination and confluence of a model transformation: in order to ensure that a model always produces the same outcome. 3) The safety and liveness properties: in order to verify that e.g. specific structural properties are preserved during the transformation. 4) The semantic equivalence of a model transformation: in order to ensure that the semantics are preserved during the model transformation.

Considering deterministic, template-based code generation, the aspects defined by Kuster can be combined into two categories – the verification of the syntax and format in general, and the semantics. The means of verifying these categories are discussed in the following section.

3.1.1 The Verification of Syntax

Different generators produce different viewpoints of the same model. For instance, one generator produces documents whereas another produces source code. Both generators interpret the input models differently as well as potentially taking into account different parts of the model. Now, in order to verify the generators, the minimum requirement is to verify that the generators take into account the desired model entities. Thus, the metamodel coverage of the generator should be checked.

In [16], an approach to verify the metamodel coverage of model transformations is discussed. The approach is based on systematically traversing the transformation rules and checking whether the rules take the all the required metamodel entities into account. The rules seem to be required to be transformation units that can be applied in any order.

Considering the actual transformation development, a model-to-model transformation development process is presented in [17]. The introduced method can be seen as a guide to capturing the requirements for the transformations in a way that facilitates the documentation and development of the transformations. Instead of capturing the transformation requirements in a natural language, the method rather makes an attempt at expressing the requirements as test cases that can be automatically validated. This approach can be seen as a test-driven development approach for transformation development.

Considering the enforcing of syntactic correctness, the metamodel should, as such, define what can be modelled, thus the input models should be syntactically correct. As regards the output, verifying the syntactic correctness is a matter of consistency checking between the output model and its metamodel [15]. In the case of code generation, the source code has to conform to the rules of the utilised programming language and the underlying software platform. When using deterministic, template-based generators, which can be considered to be the state of the practice in generator implementation, no problems should, basically, arise from issues related to syntactic correctness, termination, confluence, safety or liveness.

3.1.2 The Verification of Semantics

While the generator development process ensures that model entities are transformed as required and syntactic correctness can be verified, the semantic equivalence between the source model and the output model still needs to be verified. Maintaining semantic equivalence between the source and the target models (text can also be considered to be a model) has been studied by Kleppe and Warmer in [18]. They identify the situations in which the semantics can be preserved and when not. The situations are as follows. If neither the source nor the target model contains semantics, the semantics cannot be preserved. If the source model does contain semantics, but the target does not, then the semantics will be completely lost during the model transformation. In a case where the source model does not contain semantics, but the target does, then the semantics are not so much preserved but rather introduced. This has an interesting implication as now, via model transformation, semantics are also introduced to the source model. It must be noted that if there are multiple generators where the target model does contain semantics, then the source model contains multiple semantics depending upon the perspective from which the model is viewed, i.e. which generator is applied. The second case where semantics are preserved is when the source and the target models both contain semantics. In this case, the presumption is that the source and the target models describe the same systems.

If both the source and target models do contain semantics, some preliminary work exists to verify the semantic equivalence. In [19], an approach to comparing the semantics in the source and the target models is introduced. Both models are based on different metamodels. The approach is based on the development of a third metamodel that strives to formulate the *shared* semantics between the source and the target metamodels. The semantic equivalence is then checked by computing specific metrics across the models.

3.2 The Verification of Models

Verifying the correctness of the models means evaluating whether or not the model reflects the requirements set for an application to be developed. As models can be considered to be static when no model interpretation has been performed, e.g. by applying code generation, the verification of models means scrutinising whether all the requirements set for an application should be possible to be covered by a model simply by considering the model itself.

Maintaining a traceability link between the requirements and the model facilitates the evaluation of whether or not the model fits the requirements set for it. Gotel and Finkelstein [20] define traceability as "*the ability to describe and follow the life of a requirement, in forward and backward direction i.e. from its origins its development and specification, to its subsequent deployment and use*". This traceability link should cover both functional requirements and non-functional requirements.

The OMG Systems Modeling Language (SysML) provides modelling constructs for representing text-based requirements and relating them to other modelling elements. The SysML requirements diagram is able to depict the requirements in graphical, tabular or tree-structure format. A requirement can also appear on other diagrams to show its relationship to other modelling elements [21]

In [22], an approach to maintain a bidirectional traceability link between the requirements located in traditional requirement management tools and domain-specific models is discussed. Such an approach, when supported by tools, enables the importation of requirements to domain-specific models and the ability to connect such requirements to the corresponding design entities. This facilitates pinpointing the elements that are intended to be responsible for each requirement. Similarly, it facilitates the discovery of the requirements that are actually intended to be implemented. Considering non-functional requirements, this approach also enables maintenance of the bidirectional traceability of these kinds of requirements with measured or evaluated values from test results back to the original requirements. Considering execution time and non-functional requirements [23], the implementation has to be generated from the models and executed in order to acquire the status of the requirements. Thus, this technique also partially tests the code generator (one test case) and the underlying platform.

Models can also be analysed in situ without the need for transforming and executing the application models. The models can also be transformed into a format understood by advanced analysis tools. By analysing models, insights can be obtained regarding the behaviour of software that might otherwise only be detected the hard way. Analysis techniques can typically determine whether there are deadlocks and check that data is always delivered on time, alarm signals are always handled properly and dangerous situations are always avoided. The use of model analysis ensures that many such flaws can be avoided. State space analysis, model checking, resolution and visualisation are keywords in this area. In general, software modelling and analysis also helps to obtain a better understanding of the behaviour of the system, which often leads to cleaner, more straightforward and more versatile designs, in turn helping in mastering the enormous complexity that we see far too often in the software business [24].

3.3 The Verification of Generated Textual Representation

Verifying the GTR is about evaluating whether or not the intention of a model is realised in the outcome, i.e. in the GTR produced by a generator. The intention includes syntax and semantics. It must be noted that the intention of the model should be considered to be a fact, thus the model must be considered to be correct from the modeller's perspective in order to make any sense in the GTR verification.

Considering the intention of the source model(s) and whether or not the intention is a fit, the testing of the target model(s), e.g. an executable, should not be overlooked, although testing cannot formally state that the GTR always functions as planned. When testing is accompanied with requirement traceability, verification of the GTR should be facilitated as this method should make it possible to verify if the requirements are a fit.

In [25], an approach to monitor the quantitative, non-functional features ('ilities': aka quality attributes and extra-functional properties, etc.) of an application under development is presented. This approach is based on explicitly expressing the modeller's wish, i.e. the requirements, in application models and attaching measurement mechanisms to the corresponding parts of the models that need to be monitored. The measurement mechanisms are generated alongside the executable. The measurement mechanisms monitor the generated application during runtime and report the measured values back to the source models, therefore enabling the modeller to notice if the application performs as intended. A similar approach could also be applied when functional requirements are considered. That is, visualising the application execution in the source models as well.

To reduce the effort in developing an extensive test suite for an application, Model-Based Testing (MBT) might be worth considering, as discussed in [26][27]. MBT is a black-box testing method in which the test scripts are automatically generated from a model which describes the behaviour of the system under test [28]. The test scripts are generated from a model by utilising a set of test-design algorithms [29] that traverse the model and generate test scripts from that basis.

3.4 The Verification of Metamodels

The verification of metamodels means determining how well a metamodel encapsulates the problem domain and the rules of the target platform that are necessary to take into account in the metamodel. As a combination, a metamodel should only allow the modelling of syntactically and semantically correct applications.

To our knowledge, determining whether or not the problem domain is covered still requires neural processing, i.e. the problem domain must be carefully scrutinised and encapsulated into a metamodel. The same applies for taking the rules of the target platform into account, and there is not much we can do about it. However, guidelines on how to develop languages should facilitate the development of adequate and appropriate languages [12].

There is also an opportunity to partially verify that the metamodel only allows the modelling of correctly designed applications. 'Correctly designed' denotes that the modelled applications are syntactically correct and semantically sound in the sense that they appear to be correct from the interpreter perspective, i.e. the application models can be generated into working executables running on a set platform. A general concept for testing the entire DSM modelling infrastructure is presented in [26]. The approach consists of two phases: 1) generating a set of application models from a metamodel using an MBT approach and 2) generating a test suite for generated application models using MBT. In [27], the second phase is further elaborated and demonstrated in a laboratory experiment.

This kind of approach can be considered (according to the definitions above) to be a technique that partially verifies the metamodels. It should be noted that the metamodel is not compared to the requirements but is taken to be fact. The testing thus focuses on evaluating whether or not the metamodel is developed in such way that it prohibits the modelling of incorrectly designed application models. Applications based on the metamodel are then generated. Thus, the code generator is also partially tested in the sense that the verification also focuses on whether or not the generator produces the correct output from the input models. This approach does not strive for extensive code generator verification as all the possible paths cannot be extensively walked through in this context. When the generated applications are executed (or interpreted), the platform and the generated application are tested together. The application is tested against the use cases generated from the application models that used the MBT approach. As a result, this approach strives to test the whole DSM basic architecture.

3.5 The Validation

Validation is about evaluating if the generated deliverable satisfies the requirements that have been set for it. We see validation as the challenge of maintaining a bidirectional traceability link between the end-user requirements and the generated deliverable. In the case of DSM, such requirements should be traceable from the end-user requirements to the models, to the GTR and also back to the original requirements, but not necessarily via the models. However, maintaining a traceability link between the GTR and the models is a requirement for GTR verification. Thus, in a traceability link, the models should also be included, though this is not absolutely necessary. Some preliminary work [22] on this is already available, although there are no extensive methods in existence.

4. CONCLUSIONS

DSM is one of the most prominent paradigms of MDD, constantly showing a five- to ten-fold productivity increase in industry cases. The reason for these productivity gains is that the modelling language and the accompanied generators are specific to a highly specialized problem domain. Because of this domain specificity there are few reasons for tool vendors to emerge to maintain such specific languages or generators. Thus, the languages and generators have to be self-developed and maintained. This raises a question about the quality of the developed languages and the automation methods, which may have a negative impact on DSM adoption by the industry. Accordingly, as it is of great importance that the developed languages and generators are of high quality, there is a strong case for establishing systematic V&V techniques in order to secure the quality and appropriateness of the used tools and methods.

In this paper, V&V has been studied by adapting the definition of V&V from the IEEE Standard Computer Dictionary into the DSM context. Further, in this paper, the current means of performing V&V in the MDD context have been reported. Although the purpose was to study V&V in the DSM context, most of the work has been done in the Model-Driven Architecture (MDA) context. We found that although there are several different means available for the verification of the different layers of the DSM basic architecture, no complete method exists yet. As regards validation, maintaining a bidirectional traceability link from requirements to models to the generated deliverable and back seems to be the most prominent technique. At present there is, however, no comprehensive traceability validation method in existence.

We believe that the development of V&V methods and tools should be divided into two overlapping categories. For language developers, there should be means available to obtain knowledge about how well the metamodel reflects the problem domain. In practice, there should be means to check what kinds of undesired erroneous models can be modelled. Considering code generation, means to check what parts of the metamodel the code generator can reach would be helpful in the verification of code generators. Whether the code generator preserves semantics can be evaluated with *adequate precision* by utilizing an extensive set of semantically correct application models as an input and scrutinizing if the outcome preserves the semantics. For language users, a method and tools to automatically preserve bi-directional traceability link between all development artefacts should facilitate verification but also validation. Tools for automated test suite generation for applications would also be useful from practitioner's viewpoint.

5. REFERENCES

[1] Kelly, S., Tolvanen, J-P., Domain-Specific Modeling: enabling full code generation, Wiley, 2008.

[2] Wojciech J. D., Arisholm, E., Briand, L., , A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance, IEEE Transactions on software engineering, Vol. 34, no. 3, May/June 2008.

[3] MetaCase, Nokia Mobile Phones Case Study, 2007. URL: http://www.metacase.com/resources.html [Visited at 17.2.2010].

[4] Safa, L., The Making of User-Interface Designer, A Proprietary DSM Tool, Proc. 7th OOPSLA Workshop on Domain-Specific Modeling, 2007 www.dsmforum.org/events/DSM07/papers/safa.pdf.

[5] Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press, March, 1996.

[6] Kärnä, J., Tolvanen, J.-P., Kelly, S., Evaluating the Use of Domain-Specific Modeling in Practice, Proc. 9th OOPSLA Workshop on Domain-Specific Modeling, 2009.

[7] Fowler, M., Language Workbenches: The Killer-App for Domain Specific Languages?, URL: http://martinfowler.com/articles/languageWorkbench.html [Visited at 17.2.2010].

[8] MetaCase, URL: www.metacase.com/ [Visited at 17.2.2010].

[9] Microsoft, DSL tool, URL: http://msdn.microsoft.com/en-us/library/bb126235.aspx [Visited 16.7.2010].

[10] IBM, Jviews framework, URL: http://www-01.ibm.com/software/integration/visualization/java/ [Visited 16.7.2010].

[11] Jet Brains MPS, ULR : http://www.jetbrains.com/mps/index.html [visited 16.7.2010].

[12] Kelly, S., Pohjonen. R., Worst Practices for Domain-Specific Modeling, IEEE Software, vol. 26, no. 4, pp. 22-29, July/Aug. 2009, doi:10.1109/MS.2009.109.

[13] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 610, IEEE, 1990.

[14] Merilinna, J., Puolitaival, O-P, Menke, J., Levendovszky, T., Sprinkle, J., Karaila, M., Rencis, E., Kazato, H., Kopayashy, T., Verification and Validation in the Context of DSM group work, The 9th OOPSLA Workshop on Domain-Specific Modeling, Orlando, Florida, USA, 25-26 October 2009.

[15] Kuster, J.M., Systematic Validation of Model Transformations, in 3rd UML Workshop in Software Model Engineering (WiSME 2004).

[16] Wang, W., Kim, S-K., Carrington, D., Verifying Metamodel Coverage of Model Transformations, Australian Software Engineering Conference (ASWEC'06), April 18-21, Sydney Australia.

[17] Giner, P. and Pelechano, V., Test-Driven Development of Model Transformations, Model Driven Engineering Languages and Systems (2009), pp. 748-752.

[18] Kleppe, A., Warmer, J., Do MDA Transformations Preserve Meaning? An investigation into preserving semantics, First international workshop on Metamodeling for MDA, York, UK, November 2003.

[19] Abd-Ali, J., El Guemhioui, K., An MDA Approach to Model Comparison Based on Semantics, 10th IASTED International Conference on Software Engineering and Applications, Dallas, Texas, Nov. 13-15, 2006, pp. 393-398 (ISBN 0-88986-642-2).

[20] Gotel, O., Filkenstein A. An Analysis of the requirements traceability problem. In: Proceedings of the 1st IEEE International Conference on Requirements Engineering, pp. 94-102, 1994.

[21] OMG, OMG Systems Modeling Language V1.2, URL: http://www.omg.org/spec/SysML/1.2 [visited August 2010].

[22] Yrjönen, A., Merilinna, J., Tooling for the full traceability of non-functional requirements within model-driven development, Proceedings of the 6th ECMFA Traceability Workshop, Paris, France, 2010.

[23] Matinlassi, M. and Niemelä, E., The Impact of Maintainability on Component-based Software Systems. In: 29th Euromicro Conference (EUROMICRO'03), Turkey, 2003, pp. 25-32.

[24] Alexander, M., Gardner, W., (Ed), Process Algebra for Parallel and Distributed Processing. Chapman Hall, 2009.

[25] Merilinna, J., Räty, T., A Tooling Environment for Quality-Driven Domain-Specific Modelling, Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), 2009.

[26] Merilinna, J., Puolitaival, O-P., Pärssinen, J., Towards Model-Based Testing of Domain-Specific Modelling Languages, 8th OOPSLA Workshop on Domain-Specific Modeling. Nashville, USA, 19 - 20 Oct. 2008. Tennessee, USA (2008).

[27] Merilinna, J., Puolitaival, O-P., Using model-based testing for testing application models in the context of domain-specific modelling, The 9th OOPSLA Workshop on Domain-Specific Modeling. Orlando, FL, USA, 25-26.10.2009.

[28] Utting, M. and Legeard, B. 2006. Practical Model Based Testing: A Tools Approach, Morgan Kaufmann 1st ed., ISBN: 978-0123725011, 456p.

[29] Utting, M., Pretschner, A., Legeard, B., A Taxonomy of model-based testing, Working paper series, University of Waikato, Department of Computer Science. No. 04/2006, Hamilton, New Zealand: University of Waikato.