A Tooling Environment for Quality-Driven Domain-Specific Modelling

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

Tomi Räty
VTT Technical Research Centre of Finland
P.O. Box 1100
90571 Oulu, Finland
tomi.raty@vtt.fi

ABSTRACT

There is an increasing need for reducing costs and improving quality in software development. One of the means to reduce costs is to increase productivity by utilizing Domain-Specific Modelling (DSM). Industry cases consistently show a 5-10 fold increase in productivity when DSM is applied, in addition to a decrease of errors in generated code. In order to improve quality and especially desired quality attributes, e.g., performance and reliability, quality requirements must be considered in every development phase. Also a trace link from quality requirements definitions to implementation and tests has to be maintained to assure that the resulting application truly satisfies the requirements. As Model-Driven Development is heavily dependent on provided tool support, a tooling environment that enables quality-driven DSM would be useful. Thus in this paper, we study if MetaCase MetaEdit+ language workbench can be utilized as such by developing a code generator and a domain-specific modelling language for a laboratory case of stream-oriented computing systems. We found that the chosen environment is appropriate for an industrial application of quality-driven DSM.

Keywords

Model-Driven Development, quality attributes, traceability

1. INTRODUCTION

Whereas development costs must be abated in software development, at the same time customers demand products of ever higher quality. Today, it is not enough for software applications to satisfy demanding functional requirements. Rather, quality attributes such as the performance and reliability of an application also have to be planned, predicted, implemented and upon delivery it must attain its satisfactory and planned level of quality. Productivity must also be improved in software development to decrease development costs while achieving the desired quality.

One of the means to increase productivity is to apply modelling in software development. Model-Driven Development (MDD) treats models as first class design entities in which modelling is argued to provide a view to a complex problem and its solutions, an approach which is less risky, cheaper to develop, and easier to understand than the implementation of the actual target system [1]. In particular, the application of Domain-Specific Modelling (DSM) often results in a 5- to 10-fold increase in productivity in industrial cases in comparison to traditional practices [2].

To achieve products of desired quality, quality requirements have to be taken into account in software design and ultimately in an implementation. Much effort has been placed in developing methods and techniques that take quality requirements into account in software architecture development [3][4][5] and respectively to evaluate software quality from architectural models [6][7][8]. Whichever method is applied in architecture development where quality is of the importance, the following pieces must be employed for quality-driven development: 1) precise definitions of quality requirements, 2) a list of alternative design solutions to achieve such requirements, 3) linkage of the requirements and the design fragments that promote certain qualities, and 4) a method for utilizing such fragments in software design [9]. The preceding pieces must be accompanied with tracing of quality requirements. This is an activity of identifying requirements in the following work products through the entire development process [10]. The tracing improves all kinds of impact analysis from the measurable acceptance criteria of each quality requirement to the release of a software application [10].

To maintain a trace link all the way from quality requirements through architecture models to implementation, a link should not be broken at the model level. The models should also be automatically transformed into implementation to avoid an unnecessary phase of manually transforming models into source code. Considering the preceding statement, the subsequent requirements can be formulated for quality-driven DSM (QDSM): 1) quality requirements have to be explicitly expressed in models, 2) there must be a means to affect the quality attributes and their impact on quality should be observable in models, 3) there should be methods and techniques available to evaluate and test the models and the result of the evaluation and measurements should be presented in models to facilitate traceability of quality requirements. Finally, 4) full code generation from models has to be possible to enable testing of the models and to produce the release version of a modelled application.

Success of QDSM extensively lies in the provided tool support. Although there are tools to support quality requirements descriptions in architecture models [11], support for quality-driven development of architectures [12] and even support for architecture evaluation [11], we have not found a mature industry-ready integrated DSM tooling environment that demonstrates QDSM. Thus in this paper, we study if MetaCase MetaEdit+, which is a language workbench for developing code generators and domain-specific modelling languages (DSMLs) with a metamodelling approach, can be utilized as such an environment by developing a demonstration.

With MetaEdit+, we have developed a DSML and Python code generator which enables full code generation for simulating stream-oriented computing systems. The tool environment and the developed language provides: 1) a means to define quality requirements and to connect the requirements to corresponding model entities, 2) automated pattern recognition to provide a design rationale from the quality perspective, 3) measurement mechanisms for testing execution-time quality attributes [13] such as performance and reliability, 4) linkage between the measurement mechanisms and quality requirements to explicitly express if the quality requirements are satisfied, and 5) an optimization assistant that guides the modeller in achieving the desired qualities. We demonstrate the tool environment for modelling a system that initially does not satisfy its quality requirements, but with the help of the provided facilities of QDSM, is refined to fulfil the requirements.

This paper is structured as follows. First, requirements for QDSM including a state-of-the-art means for supporting quality-driven software development are discussed in Section 2. Second, a laboratory case including its modelling language and a code generator are introduced in Section 3. After that, in Section 4 the tool environment is demonstrated by transforming a model that does not satisfy its quality requirements into a system that completely satisfies the requirements. Discussion and conclusions close the paper.

2. Requirements for Quality-Driven Domain-Specific Modelling

The goal of QDSM is to entail in a single model the 1) quality requirements, 2) what has been done to satisfy the requirements, and 3) an evaluation and test results. By enabling this, tracing of quality requirements to implementation including test results is facilitated. Next, requirements and a state-of-the-art means for QDSM are more precisely discussed.

2.1 Expressing Quality Requirements

Quality requirements must explicitly be identified, divided and formalized to enable the designation of what parts of the application models are responsible for them and what are the means to validate the satisfaction of requirements. Considering MDD, quality requirements have to be declared not only in standalone requirements engineering tools but also in the modelling environment. Quality requirements have to be connectable to the modelling entities to maintain the explicit link between the requirements and the corresponding model entities.

There are several languages for the modelling of functional and quality requirements. These are evaluated in [14] and [15]. Also there are ontologies [16] and experimental visualization techniques [17] as well as existing tool support for requirements description, such as with IBM Rational RequisitePro and IBM Rational DOORS. Despite the format, it is of importance that the quality requirements should be defined in such way that their achievement can be verified, i.e. requirements should be measurable and they should include qualification requirements and acceptance criteria [10]. Such a template for describing quality requirements has been introduced by Ebert [10] for documenting quality goals.

2.2 Means to Affect the Quality Attributes

The utilized modelling language should have mechanisms to affect the quality attributes and there should be an enumeration of design approaches which enable to have an impact on quality attributes. It is also important that the impact of these mechanisms on the quality attributes should be made explicit to bridge the discrepancy between the quality requirements and the promoted quality attributes [18].

Patterns are considered as one of the means to express and affect the qualities of a software system. This argument is based on the definition of patterns. Alexander [19] defines patterns as "…a rule which establishes a relationship between context, a system of forces which arises in that context, and a configuration which allows these forces to resolve themselves in that context." Thus, patterns can also be seen as a solution for balancing forces related to qualities in a certain context. Currently, there is no extensive list of qualities that patterns promote. Nevertheless, some preliminary categorizations can be found from [3]. There are techniques, such as the goal-driven model transformation technique, that strives to provide a bridge between user requirements and design models [20] in which utilized patterns are based on scrutinizing the intent of patterns and dividing their intents into functional and quality parts. Then the most appropriate pattern is chosen for the situation at hand [21].

In addition to explicitly expressing the design rationale from the quality perspective, support for the modeller for the QDSM is recommended. Such support should include a model evaluation assistant that generates hints to optimize the system according to the quality requirements. Such hints can be based on e.g., architectural tactics [9] and patterns that promote the required quality requirements [3][12].

2.3 Evaluation and Testing

Models ultimately need to be evaluated and tested. There are a few software architecture evaluation methods that focus on certain quality attributes. The AEM method [7] concentrates on adaptability evaluation, whereas the IEE method [8] focuses on integrability and extensibility. There are also methods such as ATAM [6] that consider a set of quality attributes in the evaluation. While most of the evaluation methods are scenario-based or prediction methods, it is argued that quality can also be evaluated by inspecting what patterns are applied in models [12].

After evaluation, the models have to be tested by executing them to verify if the evaluation is tenable. Testing can only be performed for execution-time qualities [13] based on sheer definition. The generated implementation has to be monitored and measured from those parts in which quality requirements are connected to find out whether the execution-time quality requirements are satisfied. To support QDSM, the results of the tests need to be reported back to models. This enables the modeller to see measured values of quality attributes, and whether the quality requirements have been satisfied.

3. Domain-Specific Modelling Language for M-Net Laboratory Case

To demonstrate QDSM in MetaCase MetaEdit+, a laboratory case is utilized. The laboratory case is a stream-oriented computing system. For the laboratory case M-Net modelling language including a complete Python code generator was developed which enables complete code generation from domain-specific models. It must be noticed that the utilized laboratory case is only a laboratorial example of real stream-oriented computing system and is utilized only to simulate such a system.

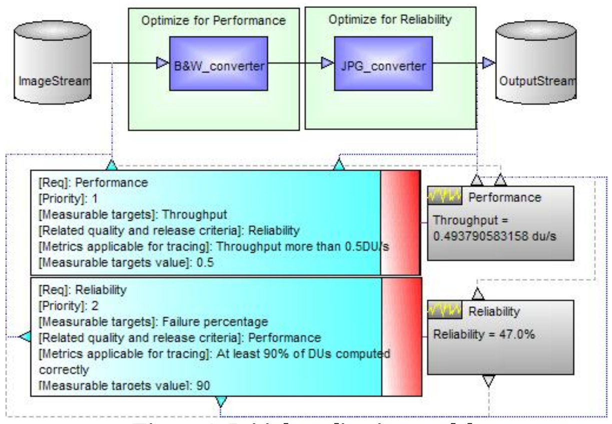
3.1 The Domain

Stream-oriented computing systems are characterized by parallel computing components that process potentially infinite sequences of data [22]. The purpose of such a system is to read data from a data source, manipulate the data and store or forward the computed data. Briefly, the system forms a pipes- and filters-based system which enables parallel processing of data. Similar systems are common, e.g., in video and image processing.

The domain of the laboratory case includes the following concepts (concepts in *italic*). *Filters* manipulate the input data and forward the filtered data to the next entity. The manipulation consumes time which is adjustable by the modeller. Computation can fail with a probability that the modeller can adjust to simulate e.g., insufficient resources during computation, program errors, or corrupted data units (DUs). If the computation fails, the modeller-definable penalty delay is endured. *Database* represents input and output pipes for the filter chain. *Switch* enables forwarding the data according to predefined principles. *Comparator* enables comparing the input data and based on predefined judgement policies, forward the input data to the next entities. *Pipes* connect various entities together.

The most relevant quality attributes are performance and reliability. Performance is the average throughput of DUs per second. Reliability is the average probability of computing and forwarding the data correctly. Reliability does have a direct impact on performance as when a *filter* fails to compute a DU, it suffers a penalty delay which has an impact on its performance.

Figure 1 represents an example of a stream-oriented computing system which consists of two *databases* and *filters*. The purpose of the application is to read a stream of colour bitmaps (from *ImageStream*) and transform the input into black and white (by *B&W_converter*) JPG images (by *JPG_converter*) and store the stream to a hard disc (*to OutputStream*).



[Req]: Performance
[Priority]: 1
[Measurable targets]: Throughput
[Related quality and release criteria]: Reliability
[Metrics applicable for tracing]: Throughput more than 0.5DU/s
[Measurable targets value]: 0.5

[Req]: Reliability
[Priority]: 2
[Measurable targets]: Failure percentage
[Related quality and release criteria]: Performance
[Metrics applicable for tracing]: At least 90% of DUs computed correctly
[Measurable targets value]: 90

Figure 1. Initial application model.

3.2 Modelling Language and Code Generators for M-Net

For the laboratory case, an M-Net modelling language was developed with MetaEdit+. In addition to the basic metamodel and model manipulation mechanisms, MetaEdit+ also provides the possibility to alter the notation of the model entities during runtime, i.e. the notation of model entities may alter depending on, e.g., properties of the entities and/or relationships between the entities. This feature becomes useful when one wants to implement runtime model validation engines. Rules for altering the notation are defined with the same language as code generators and therefore the rules can be complex as necessary.

MetaEdit+ provides a domain-specific language called MERL for developing generators. Because generators have to be developed by self, the generated code will always be as desired which might not be the case with pre-made generators provided by tool vendors. This enables complete generation in the sense that the generated output is not required to be modified afterwards. MetaEdit+ also provides an Application Programming Interface (API) which is implemented as a Web Service interface with Simple Object Access Protocol (SOAP).

3.2.1 M-Net Modelling Language

The developed metamodel for M-Net includes all concepts existing in the domain. *Filters*, *databases*, *switches* and *comparators* are the main building blocks which are connectable with *pipes*. M-Net also includes additional concepts that promote QDSM. The quality requirements are described structurally in a *requirements* model entity which can be connected to the model to cover parts that are responsible for satisfying the requirement. The template for describing quality requirements is adapted from [10]. M-Net also includes *measurement mechanisms* that enable monitoring throughput and reliability of the modelled application. The *measurement mechanisms* are connected to *pipes* in the model, i.e. similar to using probes in electrical engineering, to measure parts of the model located between the probes. The o*ptimization assistant* model entity masks parts of the model and generates optimization hints for the modeller based on quality requirements.

3.2.2 Support for Quality-Driven Domain-Specific Modelling

The developed metamodel includes a pattern recognition mechanism which evaluates the model at modelling time and automatically recognizes if the modeller has successfully modelled any known predefined pattern that promotes certain quality attributes. To render the promoted quality attributes explicitly, entities in the pattern are automatically tagged by the pattern recognition feature with a text that informs what quality attributes the pattern promotes (see text on top of *filters* in Figure 2). The modeller can instantly experience if she has managed to model an application that manifests any patterns that promote certain quality attributes.

Requirements in M-Net language utilize MetaEdit+'s notation altering mechanisms to automatically inform the modeller if the requirement is satisfied (see tags on the right hand side of *requirement* entities located bottom left in Figure 1 indicating that the requirements are not satisfied and in Figure 2 where the requirements are satisfied thus there are no such tags). The automation is enabled if *requirement* entities are connected to corresponding *measurement mechanisms* that enable measuring throughput and reliability. Values for *measurement mechanisms* are reported to model during application runtime if an application is generated in debug mode (see details in Section 3.2.3.1).

The developed language also includes a model *optimization assistant* that can be utilized for guiding the optimization of the model according to the quality requirements (see two boxes that contain the *filters* in Figure 1). The *optimization assistant* considers only entities that it contains. This enables optimizing only the required parts of the model. The model optimization assistant utilizes a generator that traverses the entities it contains and generates optimization report on that basis.

3.2.3 The Generators

Two generators that transform the models to text were developed: 1) the Python source code generator and 2) the optimization report generator. The Python code generator is utilized for transforming the model into Python source code which is not required to be modified after code generation. The optimization report generator is utilized to generate textual hints for the modeller on how to optimize parts of the system according to the desired quality requirements.

3.2.3.1 Python Source Code Generator

Filters, *comparators*, *switches* and *databases* are generated as threads thus enabling parallel processing of data. The developed M-Net Python code generator also provides the option to produce additional code for the generated application that accesses MetaEdit+'s API to animate the entities in the models when the application is executed. When this option is selected, the preceding model entities are highlighted in models when they are active during the execution. This enables the modeller to see how the system functions in real-time and also at the model level.

Measurement mechanisms are generated with the application code only when the modeller chooses to generate a debug version of the application. The value for average throughput, i.e. performance, is disclosed in the execution of the application by counting DUs passing through parts of the application that the *measurements mechanism* monitor and by dividing the count by the time that elapsed to pass the DUs forward. The value for average reliability is calculated by counting the ratio between correctly computed DUs and corrupted DUs. Counted values are reported at real-time to the corresponding *measurement mechanisms* of the model by utilizing MetaEdit+'s API.

3.2.3.2 Optimization Report Generator

The optimization report generator finds all *optimization assistants* in the currently active diagram and generates optimization hints textually according to the desired quality requirements by considering the model entities it contains. The rules for such optimizations can be very complex in real-life domains but in this simple system the rules remain straightforward and simplified. An example of a trigger for performance optimization can be formulated in natural language as follows: *"Calculate average throughput of this entity. Find all entities forwarding data to this entity and compute the sum of the throughput. If the throughput sum is more than the throughput of this entity, performance optimization for this entity should be applied."* If a trigger for optimization is fired, a textual hint is generated that guides the modeller in refining the application design. An example hint for performance optimization can be as follows: *"B&W_converter can be optimized for "Performance" by: duplicating this element and adding switches before and after these entities. See pattern: performance optimization by duplication."* As presented, a hint always contains a link to a domain-specific pattern that promotes the desired quality attributes. The pattern catalogue is included with the modelling language as pre-made example models. This enables the modeller to discover what solutions are behind the optimization and provides additional information to the modeller for him/her to make the ultimate decision whether to apply the suggested optimization. The architectural knowledge is manually coded in the optimization hint generator.

4. Model Optimization According to Desired Qualities

QDSM is demonstrated by modelling an application that satisfies its functional requirements but not quality requirements. The first attempt to model an application is then transformed into a model that satisfies both the functional and quality requirements with the aid of the provided techniques for QDSM. The purpose of the example application is to convert a stream of bitmap images to black and white JPG images. The average performance requirement is >0.5 DUs per second. The average reliability requirement, i.e. correctness of the output, is >90%. In Figure 1, the first attempt to model such an application is presented.

4.1 First Iteration

ImageStream in Figure 1 contains DUs, i.e. colour bitmaps, which are required to be computed. The time to read data from *ImageStream* is 0.1s which is defined by the modeller to mimic real-life filters. The *B&W_converter* reads DUs into its buffer and immediately after receiving the first DU it starts computing the data. Time to compute the data of the *B&W_converter* is defined by the modeller to be 2s with 100% reliability. After computing each DU one at a time it forwards the DUs to a *JPG_converter* which stores the DUs into its buffer. The *JPG_converter* consumes 1s for each DU with an average of 50% reliability. If the *JPG_converter* fails to compute the DU correctly, it is defined by the modeller to suffer a penalty of 1s. It should be noted here that the values are artificial and are only for simulation purposes.

Performance characteristics of the initial version of the application are as follows. The throughput of the system can be calculated by considering the slowest part of the system. The throughput of the *B&W_converter* is 0.5DU/s as it takes 2s to compute each DU. Throughput of the *JPG_converter* can be calculated as follows. P(*JPG_converter*) = 1/(T(*JPG_converter*) + T(*JPG_converter*_penalty)*R(*JPG_converter*)) where P is throughput, T is time and R is reliability. Thus P(*JPG_converter*) is 1/(1s+0.5*1s) = 2/3DU/s ~= 0.66DU/s. Therefore the average throughput of the system is 0.5DU/s after the first DU is computed. Reliability of the system can be calculated as follows. R(system)=R(*B&W_converter*)*R(*JPG_converter*), where R is reliability. If R(*B&W_converter*) is 1 and R(*JPG_converter*) is 0.5 then R(system) is 0.5, i.e. 50% thus the system fails to meet its reliability requirement.

The performance characteristics can also be measured by connecting the *requirements* and *measurement mechanisms* to the model and by generating a debug version of the modelled application. The *requirements* and *measurement mechanisms* are connected to the application model to cover the whole computation part of the system such as Figure 1 illustrates. By doing so the modeller can see which requirements are meant to be satisfied and which entities are responsible for the requirements.

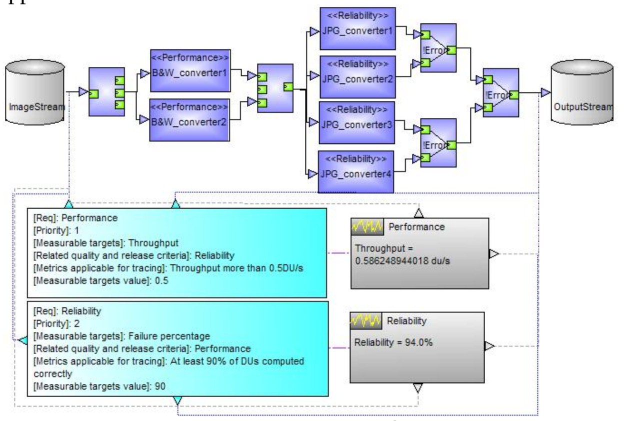
After code generation, the generated system can be executed. During run-time the system reports measurements back to the model and the requirements satisfaction indicators explicitly express if the requirements are satisfied. In this case, the system fails to meet both quality requirements (see tags on the right side of both *requirement* entities) as the average throughput is ~0.49DU/s and reliability 47%. The test was run by computing 100 DUs.

4.2 Second Iteration

The first attempt fails to satisfy the quality requirements resulting in the refinement of the application. The *B&W_converter* is the slowest part of the system and it has to be optimized for performance to increase the throughput of the application. *Optimization assistants* guide the optimization.

The *optimization assistant* enables the generation of optimization hints for the *filters* (see Section 3.2.3.2). Applying the hinted performance optimization pattern for the *B&W_converter* doubles the throughput of the optimized *filter* in an optimal case by enabling parallel processing of the data. The idea of this pattern is to forward the first input DU to the first *filter* which immediately starts processing the DU. The second DU is then forwarded to the second *filter* which starts computing the DU parallel to the first *filter*. In this way the *filters* receive every other DU and therefore halve the amount of DUs to be handled per *filter*. Thus, when the un-optimized *B&W_converter* produces 0.5DU/s, the throughput of the optimized *filter* combination is 1DU/s with 100% reliability. Now, the average throughput of the application is 0.66DU/s where the *JPG_converter* is the slowest part. The throughput requirement should be now satisfied.

Reliability of the application is still unsatisfactory therefore the *JPG_converter* has to be optimized for reliability. The *optimization assistant* produces the following hint for the *JPG_converter*: *"The JPG_converter can be optimized for "Reliability" by: duplicating this element and inputting the same data to both, and adding a comparator with the option "Error filter" after these elements. See pattern: reliability optimization by duplication."* The idea of this pattern is to compute the same data twice and forward the result to a *comparator* that forwards the corrupted DU only if both *filters* produce erroneous data. Otherwise the *comparator* forwards the first successfully computed DU. By applying this pattern the average reliability of the optimized *filter* increases, i.e. in this case reliability is increased to 75%. Applying this pattern twice results in an average of ~94% reliability. Figure 2 represents the optimized application model.



[Req]: Performance
[Priority]: 1
[Measurable targets]: Throughput
[Related quality and release criteria]: Reliability
[Metrics applicable for tracing]: Throughput more than 0.5DU/s
[Measurable targets value]: 0.5

Performance
Throughput = 0.586248944018 du/s

[Req]: Reliability
[Priority]: 2
[Measurable targets]: Failure percentage
[Related quality and release criteria]: Performance
[Metrics applicable for tracing]: At least 90% of DUs computed correctly
[Measurable targets value]: 90

Reliability
Reliability = 94.0%

Figure 2. Application model that satisfies the requirements.

In Figure 2, the application model that satisfies the both quality requirements is modelled. As can be seen, the developed pattern recognition engine automatically identifies the utilized patterns by tagging the corresponding *filters* with the promoted quality attributes. On top of the *B&W_converters* there is a <<Performance>> tag which ensures that these two *filters* participate in a pattern that promotes performance whereas in the *JPG_converters* there is <<Reliability>> tag. The tags should help the modeller to establish a design rationale for the application.

The calculated performance characteristics can be verified by generating an implementation from the model in debug mode and by executing the application. As *requirements* entities in Figure 2 does not have the tags on the right side, the application now satisfies the requirements. The average throughput measured by computing 100DUs is 0.58DU/s where reliability is 94%.

5. Discussion

We demonstrated QDSM with a laboratory-based case study of a stream-oriented computing system. For the system, an M-Net DSML and a code generator that enables full Python code generation from models was developed with MetaCase MetaEdit+. The most important means to support QDSM are based on 1) a quality requirements definition in models, 2) an automated model evaluation with pattern recognition, and 3) testing and reporting mechanisms.

Whereas a quality requirements definition technique is independent of the domain, model evaluation and testing can be considered domain-specific. In different domains, testing of execution-time quality requirements are largely dependent on what is measured and how. In this manner reusing the presented mechanisms between different domains is not possible. However, the concept remains. It is surprisingly useful to see the test results of an executed application also at the model-level. It is also useful to explicitly discover what quality requirements are satisfied. This enables to easily find out what requirements are satisfied and what parts of the model do not satisfy their quality requirements.

Design-time model evaluation for execution-time qualities was implemented in the laboratory case by identifying what patterns are utilized in the application model. As shown via the laboratory case, tentative design rationale can be obtained by utilizing pattern recognition. The evaluation could, however, also have included prediction of the performance characteristics as was done manually in the examples to provide more explicit values for quality attributes. Although evolution-time qualities such as modifiability and extensibility were not discussed in this paper, pattern recognition could also be utilized to provide some knowledge about the promoted evolution-time qualities. However, automation for quality evaluation except in the case of pattern recognition, which can provide tentative design rationale about the promoted qualities, might be a challenge for evolution-time qualities since scenario-based evaluation methods still need neural processing.

As discussed, it seems that pattern recognition can only provide tentative design rationale from quality perspective. Thus, it is questionable whether pattern recognition is sufficient for identifying the design rationale even in such a restricted area as DSM. In addition, different patterns promote different qualities and the utilized patterns might be overlapping in the application models. Therefore, finding out the design rationale by applying pattern recognition is not straightforward. In addition, sometimes it is not patterns that promote different qualities but more like functional blocks that are responsible for affecting a certain quality attribute. For instance, decreasing image resolution in image processing application certainly increases further image manipulation performance compared to utilizing high-resolution images. Decreasing image resolution might be a reason for optimizing the performance but sometimes the only reason for this is to satisfy a certain functional requirement.

As shown, by only identifying the utilized patterns it is not possible to discover the performance characteristics. Only a tentative design rationale can be obtained which, however, is still useful. Therefore to overcome the limitations with pattern recognition, next we will concentrate on manual approaches in describing design rationale by connecting requirements engineering side, where different techniques to affect the quality and their interrelated dependencies and impact to qualities are described, to the application development side. We already have developed a technique with tool support to provide measured performance characteristics from application models to requirements engineering side in order to ease the quality analysis in requirements engineering [23]. Next, we will connect the different design alternatives identified in requirements engineering to application development in a way that the impact of the utilized design alternatives to quality is automatically shown in application models. Thus, we rather strive for semiautomated approach than automated as it seems that humans cannot really be replaced by computers, yet.

6. Conclusion

There is a constant need for decreasing development costs in software development while at the same time increasing the quality of software applications. Increasing productivity can be achieved by utilizing MDD and especially DSM in software development. Nevertheless to increase the desired qualities of applications requires that not only the quality requirements must be considered at every development phase but that a continuous link from quality requirements to application design, testing and release must also be maintained. Maintaining such a link is crucial to reveal whether all the requirements set for software applications have been satisfied.

As the success of MDD extensively lies in the provided tool support, in this paper we demonstrated that there currently are mature integrated tooling environments, such as MetaCase MetaEdit+, that can be utilized as a platform for quality-driven DSM where the quality is traceable from quality requirements to application release. We demonstrated the tooling environment with a laboratory-conducted case study of a stream-oriented computing system.

7. ACKNOWLEDGMENTS

The work presented here has been developed within the MoSiS project ITEA 2 – ip06035. MoSiS is a project within the ITEA 2 – Eureka framework.

8. REFERENCES

[1] Selic, B. The Pragmatics of Model-Driven Development. IEEE Computer Society. IEEE Software, 2003, pp. 19-25.

[2] Kelly, S. and Tolvanen, J-P, Domain-Specific Modeling – Enabling full code generation, John Wiley & Sons, New Jersey, 2008, 427p., ISBN: 978-0-470-03666-2.

[3] Niemelä, E., Kalaoja, J. and Lago, P. 2005. Toward an architectural knowledge base for wireless service engineering, IEEE Transactions on Software Engineering, Vol. 31, No. 5, pp. 361-379. ISSN 0098-5589.

[4] Chung, L., Gross, D. and Yu, E., Architectural design to meet stakeholder requirements, The 1st Working IFIP Conference on Software Architecture, Kluwer Academic Publishers, San Antonio, TX, USA, 1999.

[5] Chung, L., Nixon, B.A., Yu, E. and Mylopoulus, J., Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, Boston, 2000.

[6] Kazman, R., Klein, M. and Clements, P., ATAM: Method for architecture evaluation, Carnegie Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-2000-TR-004 ESC-TR-2000-004, 2000, 83 p.

[7] Tarvainen, P., Adaptability Evaluation at Software Architecture Level. The Open Software Engineering Journal, vol. 2, Bentham Science Publishers Ltd., 2008, pp. 1-30, ISSN: 1874-107X, http://www.bentham.org/open/tosej/openaccess2.htm

[8] Henttonen, K, Matinlassi, M., Niemelä, E., Kanstren, T. Integrability and Extensibility Evaluation from Software Architecture Models – A Case Study, 2007, Open Software Engineering. Vol. 1 No. 1, pp.1-20.

[9] Bachmann, F., Bass, L., Klein, M., Moving from quality attribute requirements to architectural decisions, In: Second International Software Requirements to Architectures, STRAW'03, 2003, Portland, USA.

[10] Ebert, C., Putting requirement management into praxis: dealing with nonfunctional requirements, Information & Software Technology 40(3): 175-185, 1998.

[11] Evesti, A. 2007 Quality-oriented software architecture development, VTT Publications 636, VTT, Espoo, 2007, 79p., URL: http://www.vtt.fi/inf/pdf/publications/2007/P636.pdf

[12] Merilinna, J., Niemelä, E., A stylebase as a tool for modelling of quality-driven software architecture, In Proceedings of the Estonian Academy of Sciences Engineering. Special issue on Programming Languages and Software Tools., vol. 11, No. 4, 2005, pp. 296–312.

[13] 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.

[14] Carimo, R. A., Evaluation of UML Profile for Quality of Service from the User Perspective, Master's Thesis, Software Engineering, Thesis no: MSE-2007-03, August 2006.

[15] Etxeberria, L., Sagardui, G., Belategi, L., Modelling Variation in Quality Attributes, First International Workshop on Variability Modelling of Software-intensive Systems Limerick, Ireland — January 16–18, 2007.

[16] Savolainen, P., Niemelä, E., Savola, R., A Taxonomy of Information Security for Service-Centric Systems, Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007.

[17] Ernst N., Yu Y., Mylopoulos J., Visualizing non-functional requirements, In First International Workshop on Requirements Engineering Visualization (REV'06), Minneapolis, Minnesota, USA, 2006.

[18] Merilinna, J. and Räty, T., Bridging the Gap between the Quality Requirements and Implementation, The Fourth International Conference on Software Engineering Advances (ICSEA 2009), September 20-25, 2009 - Porto, Portugal, 6p.

[19] Alexander, C., The Timeless Way of Building, Oxford University Press, 1979.

[20] Lee, J. and Xue, N.L, Analyzing user requirements by use cases: A goal-driven approach. IEEE Software, 16 (4):92-101, July/August 1999.

[21] Fanjiang, Y-Y. and Kuo, J.Y., A Pattern-based Model Transformation Approach to Enhance Design Quality, In Proceedings of the 9th Joint Conference on Information Sciences (JCIS), 2006.

[22] StreamIt, Research overview page, URL: http://www.cag.lcs.mit.edu/streamit/shtml/research.shtml [Visited at 3.6.2009].

[23] Yrjönen, A. and Merilinna, J., Extending the NFR Framework with Measurable Non-Functional Requirements, 2nd International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, Denver, Colorado, USA, Oct 4-9, 2009.