Towards a Cognition-centric Development Method

Robert Deckers, Wan Fokkink, Patricia Lago
Vrije Universiteit Amsterdam, The Netherlands
Robert.Deckers@atomfreeit.com, {W.J.Fokkink,P.Lago}@vu.nl

Abstract. The business success factors of software development have shifted from functionality to quality (e.g., availability, security, sustainability) and development qualities (e.g., time-to-market, development cost, project predictability). However, methods and tools for domain-specific languages (DSL) and model-driven development (MDD) are still focused on providing software functionality instead of on guaranteeing quality. This paper outlines the requirements and vision of researching a method and framework that treat qualities as first class elements of a human-oriented (rather than system-oriented) development approach. An overview of preliminary results and conclusions is also included. The main subjects are unambiguous specification of quality, integrating and managing multiple quality domains, and enabling automatic transformations between specifications in those domains.
Keywords: Model-driven development, domain-specific language, software architecture, quality attribute, language theory, consistency rules.

1 Introduction

This paper explains the background and direction for researching a method and framework for integrating DSLs, making consistent specifications, and transformations between those specifications. The focus lies on the support for quality attributes (a.k.a. non-functional properties).

The vision: Software development is a human task. Software development methods must serve a process where different people, with different knowledge, capabilities, and interests, work together to achieve a result, instead of focusing on producing a machine-readable formula. A development method should support the involved people in making the right decisions, and once a (difficult and expensive) development decision has been made it must be possible to reuse it in other developments. Software development should be seen as the integration of decisions in the form of statements in appropriate languages. The ingredients and structure of such a method are the topic of this research.

The paper is structured as follows. Section 2 explains relevant issues with today's software development. Section 3 identifies solution directions for those issues. Section 4 explains the targeted result, and the preliminary results from our first studies. We end with explaining, the background, novelty, and related works.

2 People are More Important than Machines

This section elaborates on general issues with today's software development, which are a logical result of the evolution of software engineering in the last sixty years.

No language support for your own domain. Software development has started as an academic specialism where software developers performed every development task and needed to have all the required knowledge to do so. During the last sixty years, software systems have become larger and complex, as the number of stakeholders and aspects to deal with has increased. Software development has become a large-scale industry that employs people in many different areas of expertise and at multiple education levels. These people each speak their own "language" with their own idiom, and sometimes also their own grammar and syntax. This does not only hold for the development personnel, but also for customers, users, suppliers, legislators, and so on, i.e., anyone that could explicitly influence the development. The practice is however that we still use (development) languages that are an aggregation of machine primitives. They are still the main languages for development results and system specification. For example, BPMN and Java have still the same primitives as FORTRAN. The resulting disadvantage is that in many cases people must transform the mental concepts of their own knowledge into machine-oriented concepts. For example, while-loops, classes, datatypes, and inheritance are typical programming language concepts that do not come natural to most people. The forced use of machine-oriented concepts hampers people in improving and applying their domain-specific knowledge and in making domain-specific decisions. We suggest that all involved persons can use their own language and record decisions in that language. The reasoning behind this is that people make better and more reusable decisions if they can specify them in their own language [9]. *A development process must enable people to communicate and reason in their own language.*

An inconsistent view of a consistent world. Tools and methods, each with their own notations and artefacts, are already available and used for some of the domains in the development process. These include requirement management tools, project management tools, modelling tools, code checkers, or predefined runtime libraries. These tools are mostly not integrated in any tool chain, although they overlap in the concepts they use and have interdependencies. For example, a requirements engineer uses a requirement management tool and a tester uses a test tool. Although a test tool might allow you to refer to requirements, there is typically no support to guard the consistency between the requirements specification and the (related) test specifications. The same holds for example for architectural requirements and the design patterns used to realize them. Consistency is often not even considered explicitly, let alone guaranteed. *A development method must help to guard the consistency of all specifications in the development process.*

Quality matters, but is hardly covered. The ratio between machine costs and development costs has lowered tremendously in sixty years. Computers have become much more powerful and much cheaper, while the salary of programmers has followed the average inflation rate. In the early days of software development, a computer with software offered functionality that was not possible before. Nowadays, functionality of software is unique only for a short time, and soon offered by more suppliers. The business success factor of software has shifted from functionality to quality attributes like ease of use, availability, security, and sustainability, and to development qualities like time-to-market, development cost, and project predictability. In spite of that, development methods and tools are still focused on the delivery of software functionality instead of the guarantee of quality. There are hardly any methods to achieve a specified quality in an engineering way. Architecture approaches related to quality, like ATAM [17], and architectural tactics [1], systematize the process but not the specification of requirements, designs, and their relations. Moreover, the desired quality is often not specified at all. This clearly contradicts the just stated business importance of software and development quality. A development method/framework should enable the engineering of quality. This includes specifying quality and transforming specifications of quality into system specifications (e.g. source code) and development specifications (e.g. a project plan).

Needed quality is hardly specified and not clear to everyone involved. The effect is that personal beliefs and experience of the developers determine the quality of developed software. In the case of many developers with different backgrounds, this results in inconsistent quality throughout a system and its development. This is a problem especially when different developers have built different system parts. Quality cannot be proven, guaranteed, or engineered consistently, when it is not specified. *Development methods must support the engineering of quality. This includes its specification, its realization, and the trade-offs between different qualities.*

Waste of knowledge. A software system is the result of a series of development decisions. Making good decisions costs significant time and money. One must learn the involved domains and consider trade-offs between all kinds of concerns. The reuse of a good decision in another context increases the return on investment of that decision. The benefits are potentially the highest if the reuse is automated. To make a decision reusable in a useful way, it must be clear in what situations the decision is a good one, and what its effects are. Naturally, the more formal the decision specification, the more suitable it is for automation. Automated reuse of decisions can be divided in two categories.

The first category is the reuse of elements that capture the result of a decision. Those elements can be part of the working system like a GUI library, a DBMS, or a rule engine. These can also be part of any intermediate product during development, like a standard set of security requirements, process models, or design patterns. This category of decision reuse is common practice in today's software development. However, it is often unclear which decisions are reused and if they should have been reused. It is also often unclear if the integrated result is consistent. For example, func-

tional integration of two software components, each yielding certain quality properties, does not imply the quality of the integration result.

The second category motivating automated decision reuse is via transformations between development products, which reuse the decision each time the transformation is executed. Examples include code generators, compilers, or model-to-model transformations. This category is often used in current MDD approaches: a transformer turns an input specification into source code (or another intermediate format). The input model is mostly functionality-oriented and they rarely explicitly define quality. Rather, the transformer adds them each time it performs a transformation. This makes the transformer applicable only for systems that target the same quality properties. To judge the applicability of a transformer, one requires an explicit specification of both the desired quality of the system, and of the quality provided by the transformer.

The lack of awareness of development decisions that concern quality makes it difficult to explicitly exploit people's knowledge and investments in earlier developments. This is also noticeable when you want to integrate software components made by different people. The chances are high that those components do not have matching quality properties. In that case, you might want to reuse only a part of the development decisions. If all development decisions are only captured in the source code, then reusing only a part of the development decisions becomes practically impossible. *Development methods must support the reuse and automation of any decision made in a system's lifetime: from idea, to design, test, and system deprecation.*

3 Utilize and Integrate Knowledge

A method and framework based on human-thinking concepts instead of machine primitives should be the basis for a system's lifecycle that does not suffer from the described issues. Some of today's techniques like natural language processing, MDD, domain modelling, DSLs, design patterns, architectural tactics, product line architectures, IDEs (e.g., Eclipse), and collaboration systems (e.g., Sharepoint, Wiki) solve a part of the issues. But the theory to integrate them is lacking, as is explained in the following sections where we elaborate on solution direction and requirements for such a method and framework.

Integral specification of quality. There are standards like ISO25010 [16] that define a limited set of quality attributes to address quality properties. ISO25010, however, does not help either in making specifications for those properties, or in building a complete and consistent specification of the design problem, the development, or the system itself. Domain models exist for some of the quality attributes, which can serve as a basis for a DSL, e.g., the security domain model in [8]. Design patterns are available for some of the quality attributes. How to integrate specifications and design patterns for quality in a generic way is not clear yet [22]. *The research should deliver requirements and specifications to enable integration and transformation of specifications of multiple qualities.* Our research result must also contain examples of specifications of quality and explain how quality attributes can be added to the framework.

Because there is no predefined standard way to divide the world into domains, the method should ideally be applicable to any concept that a human can grasp, reason with, or specify. *The result should provide a method for specifying DSLs for different domains.* The limitations of such a method are also part of our research. Integrating DSLs is not trivial, because communication between people with different knowledge and beliefs is not trivial. Each uses his own vocabulary with own, semantics and syntax, and each thinks differently about what is important and thus what is relevant to specify.

Extend MDD to multiple domains. MDD approaches range from meta-modeling tools and meta-languages in which you build your own DSL and transformations (e.g., MetaEdit+, MPS), to more 4GL like platforms with a predefined modeling language and predefined application architecture (e.g., Mendix, Thinkwise). The models address mostly the application domain, data structure, and/or software functionality. Other system properties or stakeholder concerns are hardly modeled or even configurable. The result is that software systems built with an MDD approach have fixed and implicit quality properties. This might suit some cases, but when an MDD approach aims at a large application domain, variable quality and trade-offs between quality attributes are desirable. Also, quality should be predictable and explicitly known in many cases, for example when legislation demands it. *The research result should contain requirements and specifications of how a model of a quality can be automatically transformed into software specifications and/or software development specifications.* The result should also show examples of such a transformation.

Connect specification formalisms with human cognition. Any specification, formal or informal, can only be (partially) understood if the language in which the specification is stated, is (partially) understood. Specification languages are built on a variety of formalisms like Petri Nets, state machines, process algebra, set theory, and predicate logic. These mathematical mechanisms enable the formal specification of structures and behavior. However, to the best of our knowledge, there is no generic method that formalizes *anything* that a human can understand, reason about, and express, and that is useful for software development. For this purpose, we need to connect the formalisms to more cognition-oriented primitives. How this can be achieved is a key challenge for this research.

A useful formalism for the specification of any quality is based on any of the above-mentioned formalisms. A starting point is the Universal Grammar theory by Chomsky [5]. Chomsky's theory unifies the real world, mental concepts, and the language we use to communicate. We could say that the notion of mental concept is the smallest software particle, because any statement about something (under development) is at least based on a human notion/thought. If that notion is given semantics, then it can become a useful concept for people involved in a development process. Concept can refer to anything from the idea of a whole system, a single line of code, to a system-independent need of a specific stakeholder. Any language or specification is based on the prerequisite that at least one person is able to perceive or handle it.

This idea should be the basis of languages and methods for software development that support human reasoning and human cooperation. Hence, the title of this paper.

The method and framework should be based on formalisms for specifications that are made by a human and processed by a machine. The method must also support the gradual formalization of specification concepts.

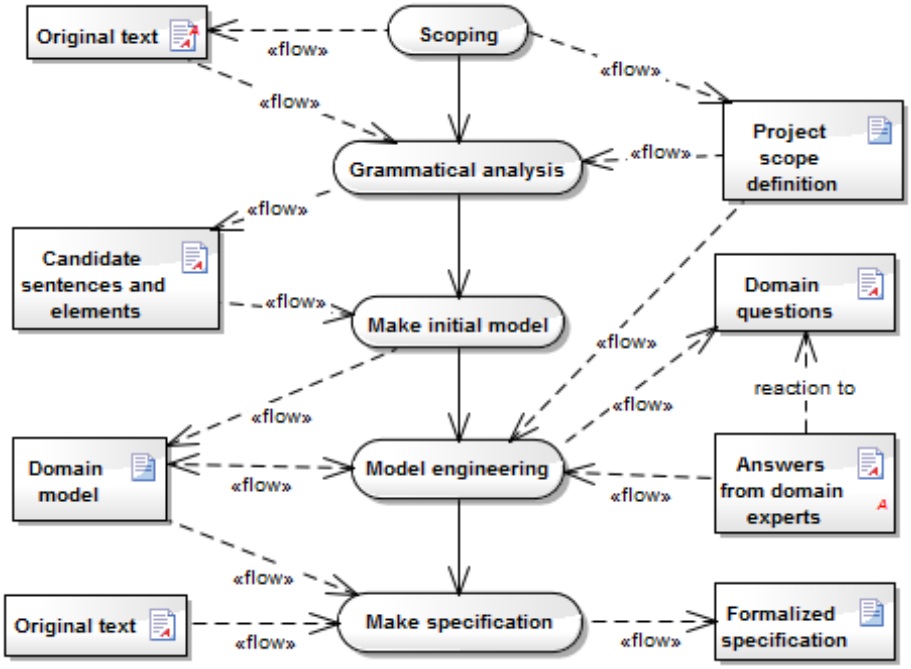
4 The Research Work and Result

The research result is threefold: (1) a method for defining DSLs, specifying quality, and their integration, (2) a framework for specifying and performing transformations between DSLs, (3) and cases where (1) and (2) are applied for a set of different quality attributes. After introducing method and framework, their contribution is shown in some examples from our cases, followed by our general observations.

4.1 A Method for Quality Requirement Formalization

From the cases and earlier experiences we have defined a method to specify quality requirements. This method is based on the KISS method for Object Orientation [20]. We adjusted the method in several ways, e.g. we replaced the KISS notation with UML and added the concept of domains. During future study cases, we will continuously refine the method.

Fig. 1. The formalization method



Our method consists of the following steps and artefacts (see **Fig. 1**):

Scoping: Define the area: which activities and objects are in scope and which are not. Define the purpose: what is the domain model used for. Select original text: identification of the used textual input.

Grammatical analysis: Extract relevant sentences from the text following a set of predefined sentence structures. Identify and define candidate objects, actions, subjects, attributes from the sentences. Eliminate homonyms and synonyms. Identify the domains based on sentence coherence and input of domain experts.

Make initial model: Create a package per domain. Fill the domain model sentence by sentence. Each package consists of three different views: the dynamic view that relates activities and classes, the static view that shows associations and generalizations, and the attribute view that specifies class attributes and activity parameters.

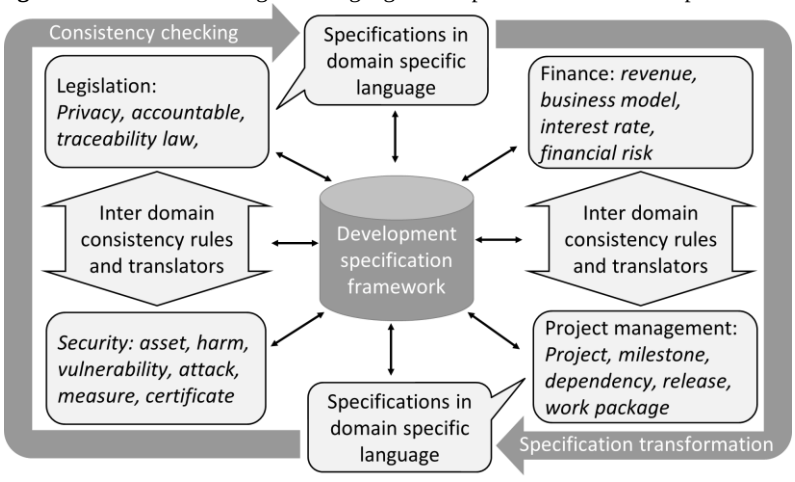
Model engineering: By iterating over the different views and performing predefined checks, the modeler normalizes the model and generates questions for the domain experts. Make object lifecycle views: for each class make an activity diagram that expresses the order that activities may be performed on objects of that class. Add domain invariants for domain properties that could not be expressed in the other modeling concepts. Continuously analyze the consistency between the views.

Make specifications based on the model and add packages for these specifications and for their relations. For a single requirement: use OCL to make a formal specification of the requirement. For a scenario: make an activity diagram that only uses activities and classes from the domain model.

4.2 The Multi-domain Specification Framework

We envision a framework (see **Fig. 2**) for specification, consistency checking, and transformation, applicable to multiple (quality) domains. We investigate existing IDEs and software engineering tools (e.g., Eclipse, UML modelling tools) on extendibility, consistency, usability, and manageability of development knowledge.

Fig. 2. The framework integrates languages and specifications for multiple domains

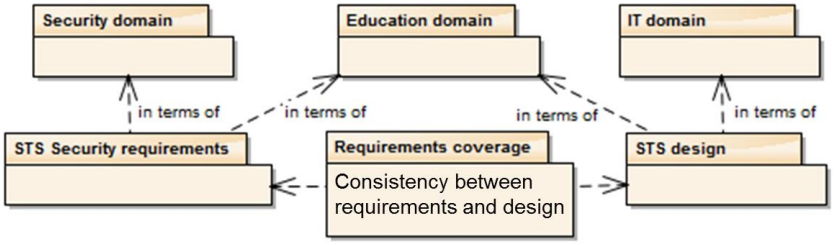


4.3 Cases

This section presents results from two of our four cases.

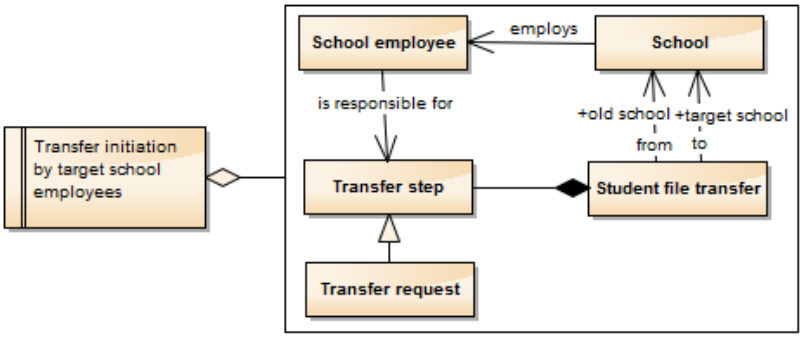
Case 1: Secure School Transfer Service (STS). When a student moves from one school to another school in the Netherlands, the STS provides secure transfer of his record to the new school. Manufacturers of Student Administration Systems (SAS) can join the STS chain by meeting an explicit set of requirements. Security plays an important role in the STS requirements. The STS requirements are detailed in several use cases and IT security measures.

Fig. 3. Security specification structure of the STS



Following the method of 4.1, we created a model (see **Fig. 3**) that covers all concepts for the specification of the STS security. There are domains for the security-related concepts, for the education domain, and for the IT domain. Following the framework of 4.2, we specify the security requirements in terms of the education domain and the security domain, and specify the STS design in terms of the education domain and the IT domain. The advantage is that the security requirements are independent from system design, and as such applicable to specific SAS designs. In future work, we will study how to make the security requirements also independent from the education domain, which will make them reusable across information systems in various application domains. The requirements coverage package specifies which IT security measures cover which security requirements. This structure offers a mechanism to specify the consistency between design (patterns) and (quality) requirements. It implements consistency specification between two domains in our framework.

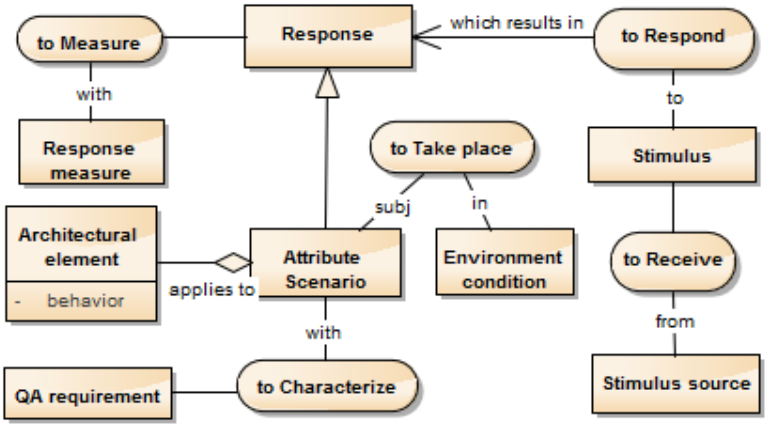
Fig. 4. Requirements context diagram in the STS security requirements package.



After establishing the domain model, we have rewritten the requirements in OCL. This way, the semantics of the domain model fully cover the semantics of the requirements. **Fig. 4** presents a part of the domain model and a formalized security requirement. For example, the original requirement "A Student file transfer can only be initiated by an employee of the target school" is rewritten as:
forAll (R :Transfer request | Exist (E: School employee, S : School | E.is responsible for.R and R.Student file transfer.to.S and S.employs.E)).
This is a formal requirements now and the consistency between the domain model and the requirement can be maintained in an automated way. How our framework should support these checks is a topic for further research.

Architectural tactics for self-adaptive software. In a research program on software sustainability, we have applied the method of 4.1 to the text in [1,27] and Chapter 4 of [1] to make a domain model that enables formal specification of quality attribute scenarios and architectural tactics for self-adaptive software.

Fig. 5. Part of the domain model for quality attribute scenarios

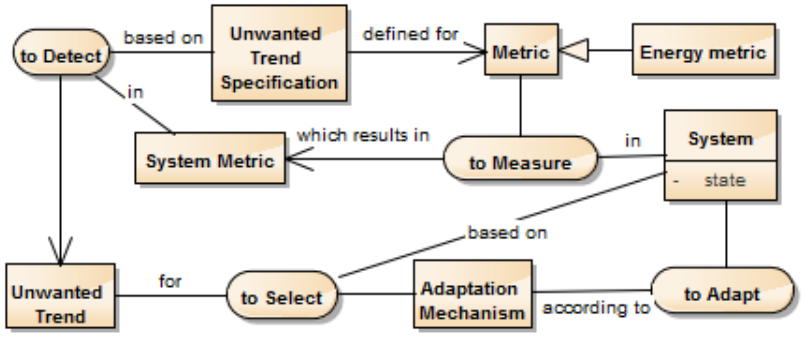


The activities and classes from **Fig. 5** form a sort of DSL for specifying attribute scenarios. A quality attribute requirement is characterized with an attribute scenario. An attribute scenario forms a response to a stimulus from a source. A scenario is applicable to an architectural element that has behavior that causes the response. Via this model, quality requirement specifications can be formalized in the form of an attribute scenario.

The domain model of **Fig. 6** is used to design self-adaptive software. It contains the activities that the self-adaptive software must perform at runtime. It refers to classes (adaptation mechanism, unwanted trend specification, metric) that are populated at design time. The model is used in two ways: (1) as a template for the elements in the operational self-adaptive software. The software must contain instances of all the classes and activities in the diagram. (2) as a DSL for the specification of a tactic for self-adaptive software, which realizes the element behavior specification of **Fig. 5**.

Currently, we are investigating how to formally relate attribute scenarios and architectural tactics via the ISO/IEC/IEEE42010 standard for architecture descriptions [12].

Fig. 6. Runtime actions in the domain self-adaptive software.



4.4 Observations

From the four study cases we carried out so far, we observe the following:

1. Formalizing quality requirements (like functional ones) is doable, but is successful only if domain experts are involved and willing to stabilize terminology.
2. In all cases, we observed that claims about quality levels are built on quicksand. When specifying quality explicitly, it becomes clear that currently system quality is mostly an assumption and based on trust in developers and testers.
3. The distinction between requirements and design disappears when considering software development not as a sequential process from problem- to solution domains, but as a set of coherent decisions taken across multiple domains.
4. Formalization of a single quality requirement often implies the formalization of several domains, because next to a model of the application domain, one also needs a model of the quality domain, and of the combination of the quality domain and the application domain. This much domain modeling requires discipline and time, but pays back in reusability.

5 Related Work and Conclusions

The idea for this research is the result of over twenty years of professional experience and education in software engineering. Experience with software and system architecture has taught that quality properties and related design decisions are stable throughout an application domain, but are hardly treated as such. Experience in MDD for business information systems, medical systems, and embedded software has shown that there is only DSL support for few people involved in a development project.

The novelty of our research lies in the combination of software architecture, DSLs, MDD, and language theory. We did not find methods that simultaneously:

- Apply models of quality properties as input for transformations into working software or other formal specifications.
- Allow the specification of any quality in a formal language.
- Provide mechanisms to extend a specification language with any domain, and for quality domains in particular.
- Put human cognition as the basis of development decisions and their specification.

The idea to integrate and unify all specifications in a development process, and let all stakeholders have their own related view, is not new, e.g., [7,8]. Business applications use several mechanisms (e.g. MVC, Data Vault) to provide users and management with consistent views on a diverse data set. We aim to apply this principle also to the specification of quality in a development process.

Software architecture literature [4,6,27] has discussed the importance of quality and the use of specific viewpoints for quality. But, a clear method on how to engineer and guarantee quality in a systematic way (via DSLs) has not been found.

Several books [1,8,11,15,17,30] present methods and examples of DSLs and MDD techniques. These examples all consider the application domain or a software design aspect. System quality is determined by transformation rules, but is not explicitly addressed. An exception is the domain model for security by Firesmith [10], which can be seen as the basis for a security DSL. In the future we plan to investigate the language elements for expressing domain models used in the specification of a particular quality and in the specification of transformations. We will also consider issues with current MDD techniques and the use of development knowledge as addressed in [9,14,20].

Well-known books on design patterns [12,13] all explain patterns in the software domain. However, they offer no traceable and consistent solution for the relation between a design pattern and its targeted quality.

Besides the required development technology, we also plan to investigate the cognitive and linguistic aspects of development decisions. These aspects seem to be neglected by the software engineering literature. As a starting point we have studied the works of Pinker on language in relation to the human mind [23,24,25,26]. So far, we did not find an obvious transition from that literature to the process of decision making and decision specification in system development.

References

1. Alizadeh Moghaddam, F., Procaccianti, G., Lewis, G. A., Lago, P. Empirical Validation of Cyber-Foraging Architectural Tactics for Surrogate Provisioning, under revision for publication, 2017
2. Bass, L., Clements, P., Kazman, R., Software architecture in practice, 3rd edition, Pearson Eduction Inc., 2013
3. Brambilla, M., Cabot, J., Wimmer M., Model-Driven Software Engineering in Practice, Morgan & Claypool, 2012
4. Buschmann et al. Pattern oriented software architecture, John Wiley & Sons Inc., 1995

5. Chomsky, N., Reflections on Language, Pantheon books, New York, 1975
6. Clements, P., et al., Documenting Software Architecture: Views and Beyond, Pearson Education Inc., 2002
7. Deckers, R., Steeghs, R., DYA|Software, architectuuraanpak voor bedrijfskritische applicaties. Sogeti, Vianen, 2010
8. Evans, E., Domain-Driven Design, Prentice hall, 2003
9. Farenhorst, R., Boer, R. de, Architectural Knowledge Management: Supporting Architects and Auditors, Vrije Universiteit Amsterdam, 2009
10. Firesmith, D., Specifying Reusable Security Requirements, in Journal of Object Technology, vol. 3, no. 1, January-February 2004, pp. 61-75
11. Fowler, M., Domain-Specific Languages, Addison-Wesley, 2010
12. Fowler, M., Patterns of Enterprise Application Architecture, Addison-Wesley, 2002
13. Gamma E., et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
14. Gonzalez-Perez, C., Henderson-Sellers, B., Metamodelling for software engineering, John Wiley & Sons Inc, 2008
15. Hemel, Z. Methods and Techniques for the Design and implementation of Domain-Specific Languages. TU Delft, 2012
16. ISO/IEC 25010, Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, 2011
17. ISO/IEC/IEEE 42010, Systems and software engineering — Architecture description, December 2011.
18. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson H., Carriere J., The Architecture Tradeoff Analysis Method, Proceedings of IEEE, ICECCS, 1998
19. Kelly, S., Tolvanen J., Domain Specific Modelling, John Wiley & Sons Inc, 2008
20. Kristen, G., Object Orientation: The KISS Method. Academic Service, 1995.
21. Malavolta I., et al. Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies, IEEE Transactions on software engineering, Vol. 36, No 1, Jan 2010, pag 119-140.
22. Me, G., Procaccianti, G., Lago, P., Challenges on the Relationship between Architectural Patterns and Quality Attributes, submitted for publication, ICSA, 2017.
23. Pinker, S., How the Mind works, W.W. Norton & Company Ltd., 1997
24. Pinker, S., The language instinct: how the mind creates language, William Morrow and Company, 1994
25. Pinker, S., The Stuff of Thought: Language as a Window into Human Nature, Penguin group, 2007
26. Pinker, S., Words and Rules: The Ingredients Of Language, Perseus Books Group, 1999
27. Procaccianti, G., Lago, P., Lewis, G. A., A Catalogue of Green Architectural Tactics for the Cloud, MESOCA, IEEE, 2014.
28. Rozanski, N., Woods, E., Software Systems Architecture, Addison-Wesley Professional, 2005
29. Stock, A. van der, et al., OWASP Application Security Verification Standard 3.0. Retrieved from http://www.owasp.org
30. Voelter, M., DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. 2013. www.dslbook.org