Åbo Akademi

OPEN ACCESS

This is an electronic reprint of the original article. This reprint may differ from the original in pagination and typographic detail.

On the Impact of Rigorous Approaches on the Quality of Development

Olszewska, Marta

Published: 01/01/2011

Link to publication

Please cite the original version:
Olszewska, M. (2011). *On the Impact of Rigorous Approaches on the Quality of Development*. Åbo Akademi University.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

This document is downloaded from the Research Information Portal of ÅAU: 18. Apr. 2024

Marta Olszewska

On the Impact of Rigorous Approaches on the Quality of Development

Turku Centre *for* Computer Science

TUCS Dissertations

No 143, December 2011

On the Impact of Rigorous Approaches on the Quality of Development

Marta Olszewska

To be presented, with the permission of the Division for Natural Sciences and Technology of Åbo Akademi University, for public criticism in the Auditorium Gamma on December 12th, 2011 at 12:00.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5A, FIN-20520 Turku, Finland

2011

Supervised by

Docent Marina Waldén
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5A, 20520 Turku
Finland

Professor Kaisa Sere
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5A, 20520 Turku
Finland

Reviewed by

Professor Mark van den Brand
Mathematics and Computer Science Department
Eindhoven University of Technology
Eindhoven, the Netherlands

Doctor of Technology Cristina Seceleanu
School of Innovation, Design and Engineering
Division of Embedded Systems
Mälardalen University
Västerås, Sweden

Opponent

Professor Mark van den Brand
Mathematics and Computer Science Department
Eindhoven University of Technology
Eindhoven, the Netherlands

ISBN 978-952-12-2671-7
ISSN 1239-1883

Painosalama Oy – Turku 2011

In loving memory of my mother, Marylka (1951-2006)

Abstract

Software systems are expanding and becoming increasingly present in everyday activities. The constantly evolving society demands that they deliver more functionality, are easy to use and work as expected. All these challenges increase the size and complexity of a system. People may not be aware of a presence of a software system, until it malfunctions or even fails to perform. The concept of being able to depend on the software is particularly significant when it comes to the critical systems. At this point quality of a system is regarded as an essential issue, since any deficiencies may lead to considerable money loss or life endangerment.

Traditional development methods may not ensure a sufficiently high level of quality. Formal methods, on the other hand, allow us to achieve a high level of rigour and can be applied to develop a complete system or only a critical part of it. Such techniques, applied during system development starting at early design stages, increase the likelihood of obtaining a system that works as required. However, formal methods are sometimes considered difficult to utilise in traditional developments. Therefore, it is important to make them more accessible and reduce the gap between the formal and traditional development methods. This thesis explores the usability of rigorous approaches by giving an insight into formal designs with the use of graphical notation. The understandability of formal modelling is increased due to a compact representation of the development and related design decisions.

The central objective of the thesis is to investigate the impact that rigorous approaches have on quality of developments. This means that it is necessary to establish certain techniques for evaluation of rigorous developments. Since we are studying various development settings and methods, specific measurement plans and a set of metrics need to be created for each setting. Our goal is to provide methods for collecting data and record evidence of the applicability of rigorous approaches. This would support the organisations in making decisions about integration of formal methods into their development processes.

It is important to control the software development, especially in its initial stages. Therefore, we focus on the specification and modelling phases, as well as related artefacts, e.g. models. These have significant influence on the quality of a final system. Since application of formal methods may increase the complexity of a system, it may impact its maintainability, and thus quality. Our goal is to leverage quality of a system via metrics and measurements, as well as generic refinement patterns, which are applied to a model and a specification. We argue

that they can facilitate the process of creating software systems, by e.g. controlling complexity and providing the modelling guidelines. Moreover, we find them as additional mechanisms for quality control and improvement, also for rigorous approaches.

The main contribution of this thesis is to provide the metrics and measurements that help in assessing the impact of rigorous approaches on developments. We establish the techniques for the evaluation of certain aspects of quality, which are based on structural, syntactical and process related characteristics of an early-stage development artefacts, i.e. specifications and models. The presented approaches are applied to various case studies. The results of the investigation are juxtaposed with the perception of domain experts. It is our aspiration to promote measurements as an indispensable part of quality control process and a strategy towards the quality improvement.

Sammanfattning

(abstract in Swedish)

Programvarusystem expanderar och blir allt mer närvarande i vårt dagliga liv. Eftersom vårt samhälle är under ständig utveckling, krävs att dessa system levererar mer funktionalitet, är lätta att använda och fungerar som väntat. Alla dessa utmaningar ökar deras storlek och komplexitet. Folk är kanske inte ens medvetna om att ett programvarusystem finns förrän det inte fungerar eller misslyckas med att utföra sina uppgifter. Speciellt när det gäller kritiska system är det särskilt viktigt att kunna lita på programvaran. På denna punkt är systemets kvalitet att betraktas som en central fråga, eftersom eventuella brister kan leda till stora finansiella förluster eller risk för liv.

Traditionella utvecklingsmetoder är ingen garanti för en tillräckligt hög kvalitet. Formella metoder, å andra sidan, ger oss möjlighet att uppnå en hög grad av stringens och kan tillämpas för att utveckla kompletta system eller bara kritiska delar av dem. Om den sortens tekniker tillämpas vid systemutveckling, med början redan tidigt i designfasen, ökar sannolikheten för att det resulterande systemet fungerar enligt givna krav. Formella metoder uppfattas dock ibland som svåra att använda i traditionell programvaruutveckling. Därför är det viktigt att göra dem mer tillgängliga och minska gapet mellan formella och traditionella utvecklingsmetoder. Denna avhandling undersöker användbarheten av rigorösa metoder genom att ge en inblick i formella konstruktioner med hjälp av grafisk notation. Begripligheten hos formell modellering förbättras tack vare av en kompakt representation av beslut som berör utveckling och design.

Det centrala målet med avhandlingen är att undersöka vilken inverkan rigorösa metoder har på kvaliteten hos det utvecklade systemet. Detta innebär att det är nödvändigt att fastställa vissa tekniker för utvärdering av rigorös utveckling. Eftersom vi studerar olika utvecklingsmiljöer och metoder, behöver vi fastställa särskilda mätplaner och en uppsättning mätmetoder behöver skapas för varje sammanhang. Vårt mål är att ge metoder för datainsamling och samla in bevis för tillämpbarheten hos rigorösa metoder. Detta kan ge organisationer stöd i beslutsfattandet om integrering av formella metoder i sina utvecklingsprocesser.

Det är viktigt att kontrollera utvecklingen av programvara, särskilt i inledningsskedet. Därför fokuserar vi på specifikations- och modelleringsfaserna, samt därtill hörande artefakter, som t.ex. modeller. Dessa har en betydande inverkan på det slutliga systemets kvalitet. Eftersom bruk av formella

metoder kan öka systemets komplexitet, kan även dess underhåll påverkas, och därmed också dess kvalitet. Vårt mål är att öka systemets kvalitet via mätmetoder och mätningar, samt även genom allmänna mönster för precisering, som tillämpas på modeller och specifikationer. Vi hävdar att de kan underlätta processen att skapa programvarusystem, t.ex. genom att kontrollera komplexitet och ge modelleringsriktlinjer. Dessutom anser vi dem vara kompletterande mekanismer för kvalitetskontroll och förbättring, även för rigorösa tillvägagångssätt.

Det viktigaste resultatet av denna avhandling är att tillhandahålla mätmetoder och mätningar som hjälper att utvärdera effekten av rigorösa tillvägagångssätt inom systemutveckling. Vi skapar och presenterar metoder för utvärdering av vissa kvalitetsaspekter, som bygger på strukturella, syntaktiska och processrelaterade egenskaper hos tidiga utvecklingsartefakter, dvs. specifikationer och modeller. De presenterade metoderna tillämpas på olika fallstudier. Undersökningens resultat jämförs med domänexperters uppfattning. Det är vår ambition att förespråka mätningar som en oersättelig del av kvalitetskontrollen och en strategi för kvalitetsförbättring.

Acknowledgements

It is a genuine pleasure for me to take this opportunity and give my most sincere and deepest credit to those, without whom this thesis and my doctoral research could not have been possible.

First and foremost, I want to express my immense gratitude to my both supervisors. I would like to thank Docent Marina Waldén, my research mother, for introducing me to the research setting and showing how the research publication should be structured and formed. She constantly encouraged me and gave me also opportunity to individually pursue challenging paths of research. I am especially grateful that she has been not only an excellent supervisor, but also a wonderful, supportive and understanding person. I would like to thank Professor Kaisa Sere for her advice and sharing her broad research experience, as well as her energetic and stimulating attitude towards research ideas. The discussions we held about my research allowed me to better motivate and strengthen my statements, and thus improved my work.

Both of my supervisors gave me a large degree of independence, placed in me immeasurable trust and faith, for which I am extremely grateful. I would also like to wholeheartedly thank them for invaluable comments to my thesis.

I am very thankful to Professor Mark van den Brand that he kindly agreed to review this thesis and that he accepted to act as the opponent at my doctoral defence. I am also grateful to Doctor of Technology Christina Seceleanu for agreeing to be a reviewer of this thesis. I would like to thank both of them for all of the time and effort they devoted to a meticulous review of this manuscript, their precious, comprehensive and constructive comments, as well as supportive remarks.

Furthermore, I would like thank my external co-authors, who provided me with their experience, domain-knowledge and diverse viewpoints. I am thankful to Colin Snook from Southampton University for supporting me at the beginning of my research with many comments and discussions on tooling issues. I am grateful for the productive and enjoyable cooperation with the Intelligent Hydraulics and Automation research group from Tampere University of Technology, in particular Matti Linjama and Mikko Huova. Thank you for giving me a completely different and innovative area to perform my research on measurements.

I would also like to thank other people with whom I collaborated with and have been friends within the Distributed Systems Laboratory: Pontus Boström for discussions and constructive cooperation, Linas Laibinis for his advice and

encouragement, as well as Dubravka Ilić and Mats Neovius for their help during my first steps in new research setting and Finnish reality. Moreover, I am very appreciative to Anton Tarasyuk, Maryam Kamali, Luigia Petre, Elena Troubitsyna, Leonidas Tsiopoulos, Fredrik Degerlund and Qaisar Malik and for being there for me, for numerous discussions and sharing all the ups and downs of my PhD studies. Although joined our group relatively recently, Yuliya Prokhorova and Sergey Ostroumov have become my friends and I am grateful for the time we have spent together.

I would also like to acknowledge colleagues outside the Distributed Systems Laboratory, namely Vladimir Rogojin, Chang Li, Bogdan Iancu, Johannes Ericsson and Torbjörn Lundkvist for numerous conversations on a variety of interesting topics. Thank you for all the great time I have had with you at work, as well as during some less formal meetings and activities. Furthermore, I am indebted to Ion Petre, Ivan Porres, Ralph and Barbro Back and Johan Lilius for their support during my PhD studies.

Moreover, I am grateful for all the fruitful and dynamic discussions within the SEMPRE group. Especially, I am thankful to Jeanette Heidenberg for sharing her valuable insight on quality of development processes, as well as for her effort in translating the abstract of this thesis. Last but not least I would like to acknowledge PhD Luka Milovanov, who has had a vast impact on the direction of my research through his software quality course.

I gratefully acknowledge Åbo Akademi University and the Turku Centre for Computer Science for excellent conditions they provided me with, as well as friendly work environment I had during my PhD studies. In particular, I would like to thank the administrative and technical personnel of the Department of Information Technologies and the administration of Turku Centre for Computer Science for their continuous assistance. In particular, I would like to thank Britt-Marie Villstrand for indispensable advice, Christel Engblom for the broad know-how on the departmental practices and Irmeli Laine for her energetic assistance in the bureaucratic matters.

The financial support of my doctoral studies and research provided to me by the Turku Centre for Computer Science, as well as Department of Information Technologies at Åbo Akademi University has been invaluable and allowed me to proceed with my work in an undisturbed way. My participation in Academy of Finland projects ITCEE and DIJON and European project DEPLOY brought not only precious collaboration, abut also funding. I am also honoured and grateful to Nokia Foundation and Finnish Foundation for Technology Promotion (TES) for granting me the research scholarships, which supported my work.

Undoubtedly, work plays an important role in my life, since it allows me to evolve and challenge myself in a variety of ways. However, all of this would not

matter if not my friends, their encouragement, motivation and ability to make my days brighter. In particular, I would like to thank Mariola, Weronika and Grzegorz Mazerscy, as well as Andrzej Mizera and Ilona Kruk for enjoyable talks and meetings. You all have a very special place in my life.

No words can express my thankfulness to my dearest family, who has always been incredibly supportive and believed in me. I am especially grateful to my parents, Marylka and Zdenek, who raised me with the passion for knowledge. Thank you for enabling me to pursue my own paths in life, creating many opportunities for me to develop and making me who I am.

Finally, I would like to thank my beloved husband for his unbelievable patience, absolute understanding and limitless love. Miki, you are the one who makes me complete.

Turku, December 2011
Marta Olszewska

List of Original Publications

[1] Marta Pląska, Marina Waldén and Colin Snook. *Documenting the Progress of the System Development.* In The Book on Methods, Models and Tools for Fault Tolerance. A. Romanovsky, M. Butler, C. Jones, and E. Troubitsyna (Eds), LNCS 5454, Springer-Verlag, Heidelberg, December 2009.

[2] Marta Olszewska (Pląska) and Marina Waldén. *Measuring the Progress of a System Development*. In the book *"Dependability and Computer Engineering: Concepts for Software-Intensive Systems"*, Luigia Petre, Kaisa Sere and Elena Troubitsyna (Eds), IGI Publishing House, July 2011.

[3] Marta Olszewska (Pląska) and Kaisa Sere. *Specification Metrics for Event-B Developments.* In Proceedings of the CONQUEST 2010, 13th International Conference on Quality Engineering in Software Technology, Dresden, Germany, September 2010.

[4] Marta Olszewska (Pląska), Mikko Huova, Marina Waldén, Kaisa Sere, Matti Linjama. *Quality Analysis of Simulink Models*. In Proceedings of the CONQUEST 2009, 12th International Conference on Quality Engineering in Software Technology, Nuremberg, Germany, September 2009. dpunkt.verlag GmbH Heidelberg, Germany.

[5] Marta Olszewska (Pląska). *Simulink-Specific Design Quality Metrics.* TUCS Technical Report number 1002, Turku (Finland), March 2011.

[6] Marta Olszewska (Pląska). *SMARTER Metrics.* Accepted to 5th World Congress on Software Quality 2011, Shanghai, China.

Contents

List of Figures

List of Tables

Part I – Overview

1 Introduction

Each day we become increasingly dependent on technology, since its presence is continuously expanding in our everyday lives. We are surrounded by software and computer systems more than ever. Even simple daily activities, such as using lifts or travelling to work involve using software that is installed in the devices that facilitate our existence.

The quality of such systems becomes an issue, especially when this characteristic can be understood differently by people with diverse backgrounds. It is a vital matter for software engineers, business managers, and researchers [51,111,81,161,116,114,69,126]. Software quality, according to the definition by IEEE Standard 1061 [54], "is the degree in which software possesses a desired combination of quality attributes". It is hard to create software [26], but it is even more difficult to achieve software of high quality [153] Moreover, it is a challenge to meaningfully evaluate the outcome [122].

Ensuring that a certain level of software quality is achieved should be assessed throughout the software life cycle by quantifying some characteristics of a system with the use of software metrics [81,46]. In particular, there is a need for software metrics and measurements for the initial stages of the development, i.e. specification and modelling [154]. These can act as supplementary mechanisms towards cost and effort savings. In general, quality measurements map the empirical world to the relational world by assigning numbers to certain attributes; their objective is to decrease bias in the evaluation of software [46]. Moreover, their goal is to control quality by giving a quantitative basis for making decisions regarding software. The application of metrics does not remove the need for human factor in software assessment through opinions and knowledge usage for analysis of measurements. From the perspective of an organisation one of the main purposes of software metrics is to have a significant effect by making software quality more apparent [54].

There are several decisive factors that impact the quality of a software product and related development process, which in consequence determine the success of the development. Firstly, it is the broad knowledge of the domain and the choice of an appropriate development methodology to be applied. Then, it is the support of suitable technologies and tools that bring the reinforcements for the system creation and evolution. Finally, there is a need for mechanisms, i.e. metrics, to assess the quality of the development.

Quality, as a broad aspect, is especially important with regard to the critical systems (also called safety-critical systems) [84]. The failure or malfunctioning

of these systems can cause some hazardous effects. For instance, it can endanger human lives by causing death or considerable injury, cause loss or serious damage to equipment, or lead to environmental harm. Moreover, severe financial losses can be involved. Therefore, creating high quality systems that we can depend on is of the essence [89].

Traditional development methods cannot assure that a high enough quality of a critical system is achieved [85]. However, this assurance can be provided by formal methods, by eliminating ambiguity and thus making the specification or a model of a system more precise. Application of formal methods brings high quality where it is needed the most, but at the same time it may cause complexity issues. Moreover, formal methods involve a mathematical background that might make people unenthusiastic about their application and sceptic about their usefulness [108]. These strengths and weaknesses of formal methods contributed to a rich history of links between industry and academia [130]. However, there is still a noticeable gap between these two communities [115], which could be diminished by tool support and evidence of applicability of formal methods. This thesis provides techniques to obtain the latter. Therefore, it is intended for both academic and industrial communities.

There is a tendency to shift the metrics from the end development stages to the phase of construction of a system [154]. The central concept is to provide empirical evidence that the application of certain techniques early in the lifecycle is a cost effective tactic towards quality improvement. This thesis presents mechanisms for the assessment of the impact of formal approaches on the quality of the initial development stages and artefacts. A suite of metrics is established as apparatus for collecting evidence that capture the information about the quality of software systems during their modelling stage. The objective is to assess and possibly increase the usability and manageability of artefacts, in particular specifications or models, developed with formal approaches. This can be achieved by complexity control and measurements that are specific to the development setting. These techniques can be applied regardless of development methodology. Hence, they can also be reused for non-critical systems.

This thesis is structured as follows. In Section 2, we present the methodologies that are relevant from the perspective of our research by describing development settings with respective tools. In Section 3, we depict quality models, as well as metrics and measurements, with special focus on complexity. In Section 4 we state generic research questions and limit the scope of the thesis by formulating the research problems in our focus. We also define success criteria and relate them to the given problems. Moreover, we depict our research process and illustrate the structure of our research. Section 5 shows an overview of papers included in this thesis. Additionally, it presents the links

between papers and defined success criteria. In is Section 6 we provide the overview of achievements, whereas related work is discussed in Section 7. We give the final remarks, together with limitations of the presented approach and future work in Section 8.

2 Development Settings

The development of critical software systems requires a high degree of rigour, which can be provided by application of formal methods throughout the development [85]. Our research is deeply rooted in the rigorous development methods, which include formal and semi-formal methods setting. By formal methods we mean mathematical techniques for developing computer-based software and hardware systems. Formal development is therefore a development that requires the application of formal methods. Semi-formal developments, on the other hand, involve the application of the formal methods only to a part of a system, be it a component or a subsystem of high criticality. They also mean omitting the proofs in favour of e.g. simulation, as main analysis technique. It should be mentioned that the semi-formal developments are also being referred to as 'lightweight formal methods' and 'semi-formal methods'. Semi-formal methods do not provide mathematical semantics behind the complete development.

The selection of the development methodology is a significant factor that influences the probability of success in the project and achieving high quality software. In our work, we have investigated formal and semi-formal approaches set in various modelling environments. The choice of these techniques was, to a large degree, determined by the projects and the case studies provided to our laboratory by either academic or industrial partners. Additionally, the degree of rigour of a method, as well as its industrial relevance, played an important role. For each of the formal methods deployed in certain setting we have established a measurement methodology and evaluation criteria specific to them. In Figure 1 we present the concept of development settings and a compact overview of the characteristics relevant with respect to our research.

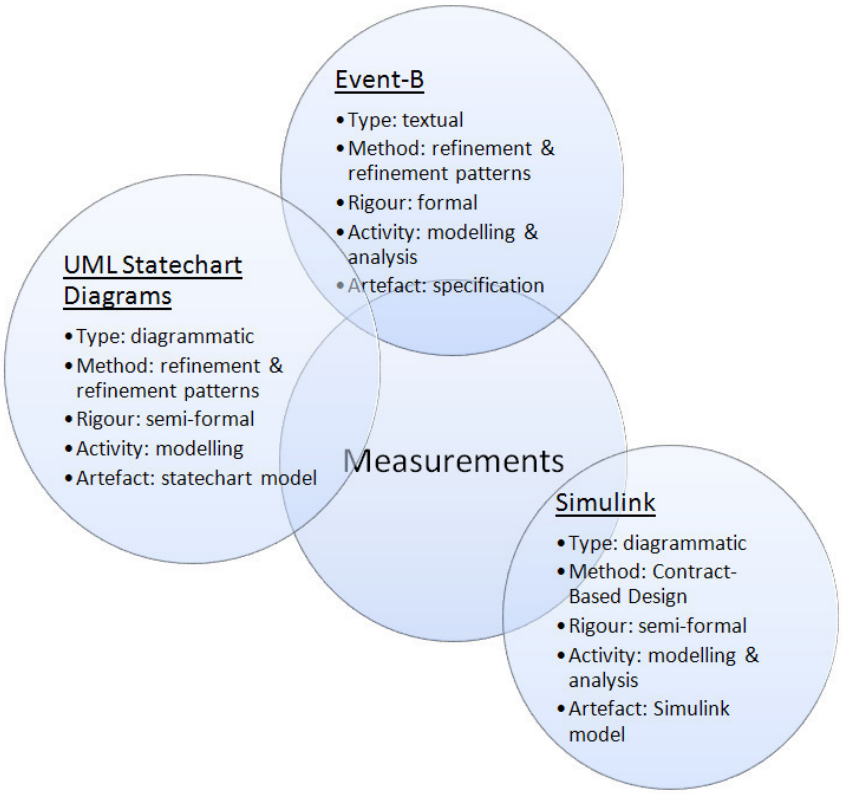


Figure 1. Development methodologies – overview

In Figure 1 we illustrate the three main development settings that we have considered: Event-B (Section 2.1), UML Statechart Diagrams (Section 2.2) and Simulink (Section 2.4). These settings are grouped around the concept of Measurements (Section 3). Furthermore, each of the settings is annotated with the type of development, the method used, the rigour it provides, the activity involved, as well as the artefact resulting from the development. UML Statechart Diagrams and Event-B are overlapping, since they share some characteristics, inter alia state machine notation; moreover, it is possible to automatically migrate from one setting and type of notation to another (described in detail Section 2.3).

In this Section, we first describe the Event-B formal method and the related refinement mechanism. Then, we present three graphical development settings: UML with statechart diagrams, the UML-B tool and the Simulink modelling environment. When depicting Simulink environment, Contract-Based Design methodology is also shown, since it provides a certain degree of rigour to the Simulink developments.

2.1 Event-B

Event-B [2,3] is a formal method and modelling language for stepwise system-level modelling and analysis, based on the Action Systems formalism [10,9,11]. It is derived from the B-Method [4], with which it has several commonalities, e.g. set-theory and refinement idea. Nevertheless, the purpose of the method application is not the same, i.e. the B-Method is focused on the development of correct by construction [38] software, while Event-B is dedicated to model full systems, including hardware, software and environment of operation [40], for instance distributed systems.

Event-B employs refinement to represent systems at different abstraction levels, which enables us to gradually introduce more details to the constructed system and to represent new levels of a system with more functionality. Mathematical proofs are used to verify consistency between refinement levels. Event-B provides rigour to the specification and design phases of the development process of (critical) systems. It is effectively supported via the Rodin platform [121], an Eclipse based tool, which is an open source "rich client platform" that is extendable with plug-ins.

An Event-B specification uses a pseudo-programming notation – Abstract Machine Notation (AMN) – and consists of a dynamic and a static part, called *machine* and *context* respectively. An abstract Event-B machine (shown in Figure 2) consists of its unique *name* and has the following constructs: *context*, which links the dynamic part with the static one, a list of distinct *variables* that give the attributes of the system; *invariants*– stating properties that the machine variables should preserve; a collection of *events* – depicting operations on the variables, where *INITIALISATION* is an event that initialises the system.

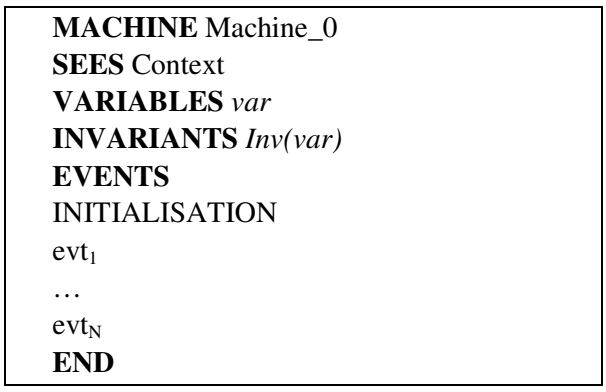


Figure 2. General form of a machine in Event-B specification

The events are specified in the form

$$\mathbf{evt_k = WHEN}\ \text{guard}_k\ \mathbf{THEN}\ \text{substitution}_k\ \mathbf{END},\ k \in [1..N]$$

where $guard_k$ is a state predicate and symbolises a boolean condition that needs to hold for the execution of the following action. A $substitution_k$ is a B statement describing how the event affects the program state and is given in the form of deterministic or nondeterministic assignments of the system variables. In case the event is parameterised it is given as **ANY** witness **WHERE** guard **THEN** substitution **END**, where *witness* is a local variable visible within an event and the guard and substitution are as before [5].

The context, shown in Figure 3, encapsulates the sets *sets* and constants *const* of the model with their properties given by axioms *axm* and theorems *th.* It is accessed by the machine through the SEES relationship.

CONTEXT Context_1
EXTENDS Context
SETS *sets*
CONSTANTS *const*
AXIOMS *axm*
THEOREMS *th*

Figure 3. General form of a context in Event-B specification

In this thesis, we first focus on the modelling of systems using the event-based approach and on developing ways of increasing the usability of the modelling activity. Then, we concentrate on syntactical properties of the Event-B specifications in order to be able to evaluate the specification with measurements.

Refinement

The Event-B method is based on a stepwise formal development method called *refinement* [38,155,8], which allows the system to be created gradually following refinement rules [103,135,152]. Stepwise refinement is a top-down approach [155], which aids handling all the implementation matters by decomposing the problems to be specified and gradually introducing details of the system to the specification. In the refinement process, an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. The refinement process is shown in Figure 4.

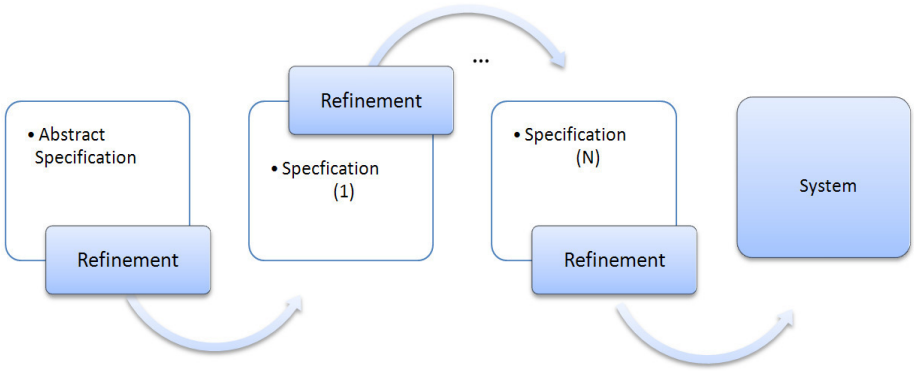


Figure 4. Refinement process

In our work, we analyse *superposition refinement* [12] and its impact on software development. Superposition refinement is a method that enables a specification to have new variables and related events that operate on them within a refinement step. This type of refinement also follows the refinement rules (referred to as *proof obligations*) [135].

The formal development starts from specifying an abstract machine (presented in Figure 2) and then refining it in a number of steps. Each consecutive machine is called REFINEMENT and is marked as such by using a separate construct in the Event-B machine (other constructs and their roles remain the same). It also identifies the machine being refined, so that the refinement chain and the modelling process can be tracked and controlled. The static part of the specification can also be refined, which is indicated by the EXTENDS clause.

The correctness of the system development, resulting in *correct by construction* [38] system, is ensured by mathematically proving that the abstract model is consistent and feasible. It involves proving that an invariant is established after the initialisation of the machine and that each event preserves the invariant.

Refinement Patterns, Decomposition and Modularity

Event-B allows the systems to be rigorously modelled relying on refinement rules. It is supported by the Rodin tool that is associated with multiple plug-ins, which together facilitate the high-level design process. In order to be able to tackle accidental complexity and to increase the modularity of large systems, various modelling mechanisms are used, like refinement patterns or decomposition methods.

In general, patterns contribute to reuse. Furthermore, they are elegant and straightforward solutions for modelling [52]. In this work, we refer to refinement patterns [135] that are specific for Event-B developments. However, other types of patterns for the Event-B setting have been investigated as well [72,73].

The decomposition techniques, on the other hand, are employed not only to reduce the accidental complexity, but also to amplify the modularity [48] of large systems. The models are decomposed and refined into several independent sub-models. This strategy allows for the proofs to be split over the resulting sub-models, which decreases the complexity of proving system properties. The Event-B decomposition tool [131] not only supports the decomposition of a model, but also allows team development over the same model. Therefore, it could add value to the large-scale developments, e.g. in an industrial environment.

2.2 UML

Event-B and its refinement method ensure the correctness of the constructed system via the respective mathematical notation. However, there are also diagrammatic modelling notations, which are independent of methods and emphasise graphical aspects of modelling, e.g. efficient communication between development team members [128]. The Unified Modelling Language (UML) [124,143] is a popular and commonly used modelling language, which supports the model-driven development (MDD) [129] and is appropriate for the top-down development framework. UML is used to model (specify, modify, construct), visualize, and simultaneously document the artefacts in the development of software-intensive systems. It is used for the representation of dynamic behaviour and static structure in a graphical manner, from different view points via different types of diagrams. This visualisation aids software engineers, managers and developers, as well as increases the understandability of the developed system. It can also serve as a common ground for the communication with customers, especially in the initial and deployment phases of the development [28,62].

In our work, we benefit from a subset of UML, namely *statechart diagrams,* since they provide a dynamic view of the system. A statechart diagram is a graph that represents a state machine. A UML state machine diagram is a behaviour diagram that is used to depict the functionality of the system by describing all possible states and state transitions of the system. The current state of the system depends on the preceding transition and its associated condition (guard). There are other diagram elements that deepen the graphical description of the system

under development, just to mention entry and exit actions, simple and composite states, as well as events.

In this work, we talk about behavioural state machines that are used to model the behaviour of specific entities. Statecharts are used to model event-driven developments (reactive systems) and describe the flow of control from one state to another state. Modelling with statecharts starts from a very general, abstract, model and is iteratively detailed with transformations to achieve a more concrete one. Among many of the intricate modelling instruments, statechart diagrams propose a method for the decomposition (for hierarchical states, also called or-states) and synchronisation (and-states). These mechanisms facilitate modelling of complex relationships between states.

A simple example of a statechart diagram is presented in Figure 5, where the statechart consists of start and final states, two states *st1* and *st2* and two named transitions *tr1* and *tr2*, where *tr1* is a self-transition.

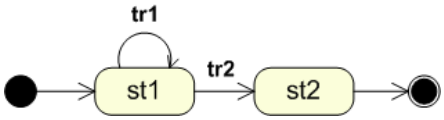


Figure 5. Example of a state machine

Modelling with the use of UML is related to modelling in an Event-B setting. Developments illustrated by UML state machine diagrams are closely linked to the ones represented by abstract state machine (ASM) notation [56,55,18]. These visualisations are provided by one of the plug-ins of the Rodin platform, namely UML-like diagrams. This type of graphical modelling is also associated with the diagrammatic form of Simulink diagrams, a graphical modelling environment (see sub-section 2.4). UML state machine diagrams overcome the restrictions of traditional finite state machines (FSM) [66,151], while maintaining their main advantages. UML state machines establish the new idea of hierarchically nested states and orthogonal regions, while expanding the concept of actions. However, due to the significantly enhanced realization of the mathematical concept of FSM, the degree of rigour is much lower [6] than in formal approaches, e.g. Event-B. Therefore, there is a trade-off between UML and formal methods. The former method is straightforward and ambiguous, whereas the latter requires a certain degree of knowledge needed for describing the system using precise mathematical notation. These factors need to be considered when deciding on the type of development process.

2.3 UML-B

A combination of quality assurance provided by formal methods and the intuitiveness given by graphical modelling notations can be found in the UML-B tool [133,134,144]. It is a graphical front-end to Event-B, which enables the visualisation of the system being modelled. UML-B narrows the existing gap between formal methods research and practical software development [115], by integrating the formal reasoning with the UML-like constructs. It uses diagrammatic notation based on UML [113] style, i.e. state machines, which increases the understandability of the model. The visualisation of the system modelling increases the usability and user friendliness of formal approaches by improving the understandability of the development [120].

UML-B offers the functionality for drawing state machine (and class) diagrams, and translating them directly into Event-B. The strong integration with Event-B tools makes the Event-B static checker and prover to automatically carry out the verification of a model, so that the errors found at this stage are apparent on UML-B diagrams. UML-B supports refinement mechanisms and fits well with the Event-B refinement framework. UML-B is an open-source tool, which uses the Eclipse Modelling Framework (EMF) [140] to generate a repository for UML-B models from a meta-model diagram. The drawing tool is based on the Graphical Modelling Framework (GMF) [53].

2.4 Simulink

Simulink® [132] is another diagrammatic modelling setting, which is used in our work. It is a MathWorks [92] commercial toolbox and an environment for Model-Based Design [110,127] of dynamic and embedded systems that gained industrial importance. It provides an interactive graphical environment and a customisable set of block libraries that allows one to design, simulate, implement, and test a variety of time-varying systems, including communications and controls. Simulink can be used to examine the behaviour of a variety of real-world dynamic systems, which allows multi-domain simulation of the created models.

Simulink Diagrams

An example of a Simulink model is presented in Figure 6. The dataflow diagram is made of rectangular blocks that consist of one or more inputs, states and outputs. They are inter-connected by arrows, called signals, which represent connections of block inputs to block outputs. Each block represents an elementary dynamic system that produces some output. The elliptic shaped

elements in the diagram are called inports and outports and represent the connection points between blocks and signals.

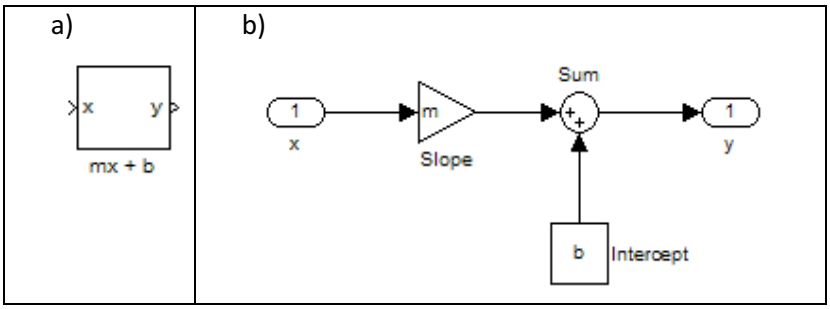


Figure 6. Subsystem block (a) with its contents (b) (example taken from [96])

The blocks in the model can be structured into subsystem blocks, which facilitate modelling and allows for the models to be hierarchically structured, as seen in Figure 6. Such subsystem blocks can have any number of ports for input and output of data to and from the system, respectively. From the measurements point of view we are interested in the structural aspect of the diagram and interrelation between the subsystems of the Simulink model. We are also addressing the issue of the impact of a rigorous development methodology on the quality of the model.

Contract-Based Design

In order for the systems to be built in a correct by construction manner, the Contract-Based Design method has been used for Simulink [19,21]. It is a rigorous approach for stepwise design that incorporates modular techniques of system design with formal reasoning about their correctness. Contracts consist of the pre- and post-conditions for programs or their parts. They give directives for decomposing functionality into components, as well as insight on the analysis of system correctness. The idea of contracts appeared in [109,105,106,107] giving clear guidelines for the design process, covering the inheritance and exception handling issues, as well as documentation. Contracts in Simulink provide a comprehensive formal background for the design and analysis of systems in a setting that is industrially popular and powerful. Moreover, contracts help to construct the system by providing a certain degree of control over development, thus limiting the number of defects introduced into the system. Contracts also facilitate the documentation of the system during the design.

3 Quality Measurements and Metrics

High *quality* of software is considered essential for the critical systems [89]. Therefore, quality assessment of artefacts related to critical systems is important regardless of the development setting and type of rigorous methods used, or the application domain. Software quality is the degree in which the given software has a grouping of quality characteristics that is considered necessary [54].

Software quality can be assessed using measurements and metrics. Measurement is the process of assigning numbers or symbols to properties of an object in the real world in such manner that they are described according to clearly defined rules. Thus, measurement can be interpreted as a direct quantification of a property, while a measure is the number or symbol assigned to characterise the property of an object. Metrics, on the other hand, provide techniques that transform the data and represent relations between certain characteristics [46].

Since we are interested in accurately investigating the impact of rigorous approaches on the quality of developments, we need to clearly state the goals of our study. We achieve this by identifying our high-level quality objective, the development settings, the characteristics we are focusing on, the target group that might find our results useful, as well as the artefacts that are examined.

Our objective is to assess and analyse the impact of rigorous types of development methods on the quality of produced artefacts, rather than rigorously model some large-scale and complex critical system. The formal methods in our interest are Event-B and Contract-Based Design, while statecharts are in our focus for semi-formal developments. The secondary aim is to increase the maintainability, usability and improve the understandability [138] of the formal designs with the use of visual notations, as well as setting-specific metrics and measurements.

The challenge is to tackle the formal modelling issue from the perspective of the developer and manager. It is addressed by facilitating the development process in its early stage, e.g. by providing control of design with patterns and quality measurements.

The work presented in this thesis concerns evaluation of certain artefacts: a specification, a model of a system or component, impact of a development methodology on the artefact or the development process itself. In order to assess these, it was necessary to provide suitable evaluation techniques. In our research, we have mainly relied on the direct and indirect measurements specific for the development setting. The latter were computed according to the metrics we established.

Our metrics are derived from the ones recognised and commonly used in software engineering community. We have adapted them for different development setting and domain. These metrics are considered as artefacts in the following parts of the thesis. Knowing the history of application of certain metrics, their advantages and drawbacks, we have been able to narrow down the feasibility of metrics to our purpose. The measurements include product and process measurements. Our objective is to evaluate (software) system quality at the early development lifecycle with respect to maintainability and usability, as well as characteristics related to these.

The maintainability and usability characteristics are strongly and directly affected by complexity [60]. In general, the complexity and size of the systems are continuously growing in every application field due to increasing requirements that need to be fulfilled. The *essential complexity* is the lower bound of the degree of complexity of the system after which the complexity of the system can only escalate, possibly leading to worse quality. It is the *accidental complexity* that can and should be controlled [33], in order to obtain high quality products. Therefore, we are interested in investigating complexity, especially in the design stage, in diverse modelling development settings, where the control and feedback provided for the project can help in achieving successful development.

It needs to be mentioned that we have intentionally concentrated on the engineering aspect of the proposed solutions. This practicality in our work originates from the fact that measurement as such is a hands-on undertaking and needs to be usable and applicable.

This Section is structured as follows. First we present the general concept of software quality and then illustrate the idea of quality models. Next we describe the maintainability and usability attributes. Finally, we show the dependability aspect in software quality research.

3.1 Software Quality

It is beneficial to have clear quality goals when developing a quality software product. The quality characteristics, called *attributes*, and their relationships should be defined at the start of the development. There are two types of attributes: *internal* and *external*. The former are being measured purely in terms of the artefact itself, whereas the latter are dependant on other factors, e.g. setting and human comprehension. We describe them in more detail in Section 3.2.

ISO/IEC 9126, which is now relabelled to ISO 25000 series, gives a clear decomposition of external quality attributes [138], where the quality consists of

six attributes: functionality, reliability, usability, efficiency, maintainability and portability (see Figure 7). One should mention that these signify aspects of end-product quality for the software to be developed [27]. Some attributes are overlapping with those given in IEEE 1061 standard [54].

A different standard, e.g. ISO/IEC 15504, can be used for the assessment and improvement of development process [139]. In our work, we study the impact of used development methodologies on the quality of product and related artefacts, such as specification or model. The observation of a development process is thus an inevitable part of our research.

Figure 7. The ISO / IEC 9126 quality attributes

Quality Models

A systematic representation of features that are important for the developed system allows for more disciplined evaluation of software quality. Therefore, the notion of quality is often represented as a *model*. There are many *fixed software quality models*, just to mention Boehm [17] and McCall [100] models. They use a tree-like approach, see Figure 8 [46], where high level attributes identified as *quality factors* are iteratively decomposed to lower level sub-attributes, called *quality criteria*. The leaf level of this tree consists of *quality metrics*, which enable measurements of a specific criterion. The early quality models [17,100] are to some degree included in above mentioned ISO/IEC 9126 standard.

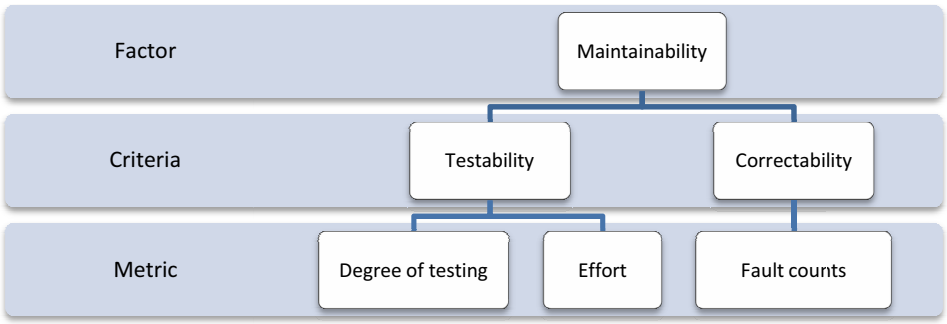


Figure 8. Example of a decomposition tree of maintainability attribute

Software quality models and their derivatives can be customised for individual purpose [42] by accepting the quality decomposition concept and focussing on the quality attributes of the highest priority for the given development. The *customised quality models* can also give the key software attributes for a certain development phase or development artefact, e.g. specification.

For the purpose of our research, we defined our own software quality model according to the priorities of investigated developments. The model consists of *maintainability* and *usability* attributes. We concentrate on these from the perspective of early stage development artefacts, i.e. a specification and a software model, rather than a deployed product. Early control of development allows identifying problems in the initial stages. Timely reaction to these issues is cost efficient and requires less effort, in contrast with their discovery in later development stages [136].

In Figure 9 we present the quality attributes (factors) with their sub-attributes (criteria), which are the measurement objectives in our research. It should be mentioned that some quality factors are not independent, meaning that they can have overlapping criteria and be characterised using the same metric.

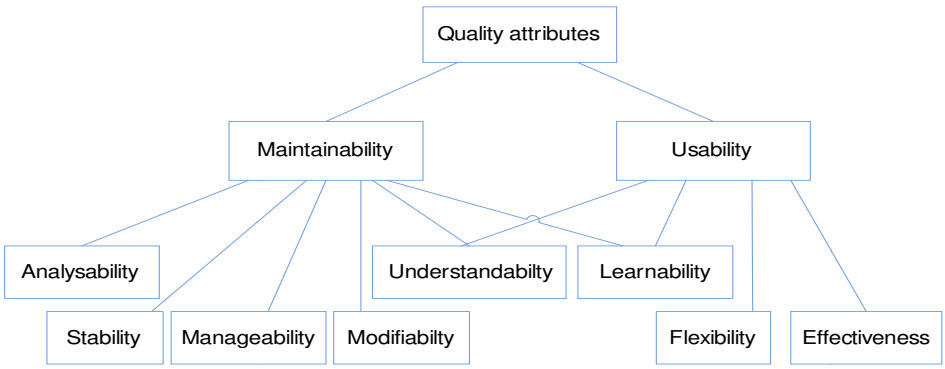


Figure 9. Quality attributes in our research

Maintainability and Usability

We define *maintainability* as the capability of an artefact to undergo repair and evolution. We investigate issues related to the maintainability attribute, meaning the easiness to analyse, manage and modify an artefact, e.g. a specification or a model. We also explore the stability issue with respect to certain type of data-flow models. Here, we do not discuss the entire software that is executable or deployed; rather, we concentrate on the system specification and design stage by evaluating aspects of interest for each attribute. We share the opinion of the Software Engineering Institute [14] that the techniques for the assessment of a system design before it is built have a great value. We focus on analysability, modifiability, stability and also manageability sub-attributes, which can be found well structured in the ISO/IEC 9126-1 standard (superseded by ISO/IEC 25000: Software engineering: Software product Quality Requirements and Evaluation, SQuaRE, Guide to SQuaRE) [138]. To some degree reusability of an artefact is also in the scope of our research.

We also examine the *usability* attribute defined as a measure of how well users can benefit from some artefact, be it a model of a system or some of its components. Usability and its sub-attributes are evaluated for the artefacts at the early stages of development. We focus on these human related sub-attributes, i.e. understandability and learnability with respect to a specification and a model of software system. Moreover, we investigate flexibility and effectiveness of formal methods, since the development approach is often decisive when constructing software. Above mentioned sub-attributes are also important for the assessment of the suitability of the development methodology and possible improvements of the development process. It should be mentioned that usability is treated by ISO standard on the same hierarchy level as maintainability. We concentrate on maintainability and usability in order to use our findings to influence the attitude of developers and managers towards formal approaches.

Dependability Aspect

Usability, as well as maintainability (more precisely, its sub-attributes adaptability and manageability), directly affect the *dependability* property [7]. For safety critical systems it is dependability that is the key property, understood as ability to avoid service failures that are more frequent and more severe than is acceptable [7]. It needs to be mentioned that the notion of dependability has been recognised by various communities in a different way. We use the dependability taxonomy first presented by Laprie [86], then Software Engineering Institute (SEI) [14] and then confirmed and further extended by the dependability experts [7]. In Figure 10 [7] we show a viewpoint on structuring

of the quality attributes that is different than the one presented earlier in this section. The Dependability property is decomposed into six quality sub-attributes: availability, reliability, safety, confidentiality, integrity and maintainability. We focus on the very last one, as we previously described.

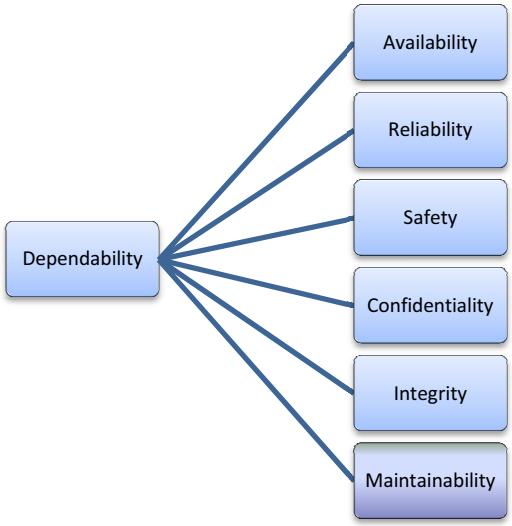


Figure 10. Dependability attributes

After we decompose the quality attributes, we have to decide on specific measures for the lowest level attributes. Subsequently we need to find the relationships between the measures. We focus on complexity, as it is a characteristic that impacts both maintainability and usability. It is influenced by the type of the development method and specificity of the development setting. Additionally, we are investigating the problem of (generic) refinement patterns and modularisation of models in the perspective of maintainability and usability attributes.

3.2 Measurements and Metrics

Measurements and *metrics*, as well as *analysis* of the measurement outcome are vital in the assessment of the quality of the system or development method [46]. They are applied to the lowest level sub-attributes in the software quality model. Metrics provide techniques that transform the data and represent relations between certain characteristics while measurements give "facts and numbers" type of answers about measured artefacts. Measurement should be *effective* [43], i.e. carefully planned and targeting the clearly defined goals among other organisation-specific criteria. *Analysis* is used to bring understandability aspect to measurement activity, as well as to present the findings. Another important

concept in the measurement terminology and decision making is an *indicator*. It relates a metric to a baseline or an expected result and provides an early insight into development and its quality.

We collect direct and indirect measurements [45] specific for certain development setting, which regard *product* and *process* measurements. We record size measurements, primitives count, development duration, duration of certain development phases, number of defects, number of elements that are setting-specific, all of which are considered as *direct* measurements. The *indirect* measurements are obtained with the use of metrics and measurement models that are regarding the relations between the direct measurements, e.g. number of defects with respect to their origin and removal phases, interrelations between elements, etc.

We mostly focus on internal (static) attributes, meaning that they are measured entirely in terms of the products or processes themselves and do not rely on software execution [138]. The external attributes are considered when human factors are involved, e.g. regarding the perception of the developers. We mostly base our quality analysis on the *quantitative* data, which means that in order to describe some attribute we use vast range of numerical measurements. However, we validate our results with experts from the domain, who provide us with *qualitative*, even subjective, assessments via non-numerical measurement methods.

All of the measurements should be *meaningful*, meaning that they should preserve their truth or falsity regardless of the change under allowable transformation [42,44]. Meaningfulness also enables us to establish the type of operations that can be performed on various measures. For instance, it seems that the meaningful measure of complexity as a generic feature is impossible to achieve, however a specific view of complexity is not considered as a "holy grail" anymore [42]. The viewpoint and assumptions are necessary to use the measures in a proper and relevant manner [163].

3.3 Complexity

Complexity is a characteristic that impacts quality of the software system. It influences not only reliability due to higher probability of error occurrences [159], but also maintainability [13], and in particular understandability of the system and later system reuse. Moreover, it affects development effort, costs and risks [30]. Since the complexity of the software (systems) is continuously growing [26,148] and is projected to grow geometrically [137], we focus our interest on establishing complexity models for certain development settings. In order to manage the complexity, it is necessary to have suitable means of

measurement and analysis. We find it beneficial to measure complexity and related features already at the early development stages. Our work involves:

- syntactical complexity for the analysis of complexity on the language level (Halstead [58]),
- system complexity for the inter and intra complexity analysis on the model level (Card and Glass [29])
- cyclomatic (conditional) complexity for the analysis of independent paths in a program (McCabe) [99]
- design quality metrics for the analysis of the dependencies within the model (Robert C. Martin [91]).

Complexity in software systems can be assessed from many perspectives, depending on the development setting or granularity of the investigation. We address the problem of specifying the indicators of complexity in the syntax of specification language and in a model of a system at the design stage. Here, we present in detail the measures and metrics that we listed earlier.

Halstead's Software Science

We benefit from the controversial *Halstead's Software Science* [58] and derive syntactical metrics for Event-B specifications from Halstead metrics. The original metrics are based on a collection of tokens classified either as operators or operands. The number of different tokens (n1 for operators, n2 for operands) and the total number of occurrences (N1 and N2, respectively) of each token are calculated. Based on these primitives, a system of equations was developed. It expresses the total vocabulary *n*, the overall program length *N*, the actual volume *V*, the program difficulty *D*, the program level (which is commonly considered as a measure of software complexity) and other features, like the development effort *E*. The equations are as follows:

i. $n = n_1 + n_2$ (Vocabulary)
ii. $N = N_1 + N_2$ (Length)
iii. $V = N * \log_2(n)$ (Volume)
iv. $D = (n_1/2) * (N_2/n_2)$ (Difficulty)
v. $L = 1/D$ (Level)
vi. $E = V/L$ (Effort)

We are aware that there have been many critical opinions about these proposed metrics [59], e.g. the difficulty of deciding whether a token is interpreted as operator or operand can be mentioned here. Moreover, the assumptions of Halstead metrics that regard effort estimations seem theoretically dubious. Finally, there are not enough studies to confirm or reject the validity of

these metrics. However, we carefully adjusted the metrics to the Event-B setting by meaningfully defining primitives with regard to the Event-B dynamic and static parts of the specification, as will be described in Section 5, Paper 3. Furthermore, we included the refinement mechanisms and their impact on certain specification constructs to our model.

Card and Glass Complexity

The *Card and Glass complexity metric* [29] is a system-level complexity model and is represented as a sum of *structural* and *data complexities*. The metric is based on the structure of the model of the system and its interrelations, as well as Input-Output properties. *Structural complexity S* is defined as the mean of squared values of fan-out per number of modules:

$$S = \frac{\sum_i f^2(i)}{n},$$

where *f(i)* is fan-out of module *i* and *n* is a number of modules in the system. Fan-out is a count of modules that are called by a given module.

Data complexity D is defined as a function that is dependent on the number of Input/Output variables and inversely dependent on the number of fan-out in the module. This is given by the following equation:

$$D = \frac{V(i)}{n \cdot (f(i)+1)},$$

where *V(i)* is the number of Input/Output variables in a module *i*, *f(i)* and *n* are as above. The total complexity *C* is computed as a sum of structural and data complexities presented earlier (*C*=*S*+*D*). Card and Glass complexity model gives guidelines on accomplishing a low complexity design. In our work, it is used in the Simulink modelling environment, with respect to the elements specific to the Simulink diagram, as will be presented in Section 5, Paper 4.

McCabe Complexity

McCabe complexity [99] is a software metric representing the number of linearly independent paths that comprise the program. It is a cyclomatic number, which is computed using the control flow graph of the program: nodes and directed edges that connect these nodes. The complexity is computed according to the formula:

$$C(G) = e - n + 2p,$$

where C is the cyclomatic complexity of graph G, e and n is the number of edges and nodes of the graph G, and p is the number of connected components. The

measurement of cyclomatic complexity was designed to indicate testability and understandability (maintainability) of a program. In our work, we apply it to Progress Diagrams and indirectly to Event-B graphical representation. We also extend the applicability of this metric to statechart diagrams. It is presented in Section 5, Paper 2.

Martin's Object-Oriented Design Quality Metrics

We also benefit from *Martin's Object-Oriented Interdependencies measure* [91], which presents a set of metrics that can be used to evaluate the quality of an object-oriented design from the perspective of the inter-relations between the subsystems of this design. The proposed metrics measure the degree of correspondence between the design and the pattern of dependency and abstraction, which were defined by the author as sound with regard to his criteria. We adjust these metrics and apply them in the Simulink environment in order to assess the model from the perspective of maintainability and reuse, as well as way to indicate the possible fragilities in the design. This is described in Section 5, Paper 5.

4 Research Questions and Research Process

Our research was inspired by the emerging problems and requirements presented by the ongoing projects in the Distributed Systems Laboratory, specifically RODIN (Rigorous Open Development Environment for Complex Systems) and its continuation DEPLOY (Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity), as well as ITCEE (Improving Transient Control and Energy Efficiency by Digital Hydraulics). The possibility of experimentation within these projects was an important factor in the research process, e.g. when combining the existing methodologies or using them in a different context. The research results facilitated the cross-domain technology transfer and contributed to the overall outcome of the projects.

Our work has also been determined by the current status of the research on measurements performed for critical systems, as well as the need for evidence of the impact of rigorous methods on the system developments. Furthermore, the open research possibilities were identified as a motivating force that guided to the innovative aspects of this work.

In this Section we first characterise the generic questions that this thesis addresses by identifying the objective of our work and decomposing it according to the research setting. Then we specify the research problems and challenges. Next we define criteria, which we denote as a successful result of our research. This is followed by the description of the research process, i.e. methods and techniques used, as well as the illustration of the research structure.

4.1 Problem Characterisation

Formal methods have proven to be successful in industry in the safety critical applications [31,35,125]. The example often mentioned is from the transportation domain is the Meteor line 14 driverless metro in Paris [39,24]. However, there are several other large-scale success stories, the outcome of which can be observed in every day life [87,79,16]. The described triumph of formal methods gives the impression of promising results when integrated into the development process. However, some reservation remains regarding the (industrial) application of formal methods such as: the degree of feasibility of the method for the application area, cost and time necessary for technology transfer, impact of the methodology on the development duration, etc [47]. Issues such as the mathematical background of a developer and a feasible tool support are identified as the obstacles for the technology transfer.

Therefore, evidence should be collected to increase awareness regarding the rigorous approaches. The need for appropriate measurement system for "validating the claims of the formal methods community that their models and theories enhance the quality of software products and improve the cost-effectiveness of software processes" is already acknowledged in [150,44]. The advantage of formal approaches in industry from the perspective of effectiveness of the methods used is questioned by some researchers in [34,157], where the authors point towards lack of accurate and scientifically based measurement data.

We tackle the problems with lack of evidence attesting that formal methods positively influence the quality of the developed systems. We search for facts, which could be the decisive factors to provide insight to the managers and other industry representatives about the potential that formal methods bring. The benefits and, perhaps, drawbacks of the approach need to be demonstrated via "facts and numbers", so that conscious decisions regarding the system development methodology can be made.

The evidence collection regarding the impact of formal methods on the development process and product quality implicates providing particular instruments for the assessment. This means setting up a measurement program or providing a set of metrics specific for certain development setting. The measurement program should encompass a collection of metrics that accurately reflect the attributes about which we want to have the information. For instance a top-down *Goal Question Metric* (GQM) approach [15,149] or *GQM+Strategies* [90] can be useful for identifying the measurement objective.

One should keep in mind that the focus of quality measurements has been shifted from end-development phases to the early stages, e.g. to specification and modelling activities. This means that if already existing and validated metrics are reused, they need to be "lifted" and adjusted to fit the current purpose. Afterwards, they need to be validated against a scientifically significant case study. Moreover, the experiments should be repeated, so that the observations made are sound and robust.

Until now, there have been either some quality assurance activities or measurement programs that have been imposed by the stakeholders or certain standards. Many of these could not be published due to confidentiality matters. We consider the confidentiality issues and privacy policy regarding the (large-scale industrial) projects as the most important obstacles for data collection. We do understand the reasoning behind the secrecy; however, we strongly feel that these barriers are the main causes for the impediment in the information flow. Hence, the statistics and the access to the pilot developments given by the

industrial partners within the running projects are crucial when it comes to the research in the measurement area.

The fundamental driving forces of the research are the unsophisticated means of evaluation of the systems at the initial phases of their construction or the absolute lack of these. It is proven that introducing changes, detecting and fixing defects, as well as making adjustments in the early stages of the software (system) life-cycle are cost-effective [123]. Therefore, we want to provide techniques for the assessment of product and process artefacts in developments on a high, more abstract level.

Main Objective

The main objective of this work is to measure the impact of rigorous approaches on the development of software systems. The goal is to provide methods that enable us to assess the quality of the product, be it specification or model, created with formal or semi-formal techniques. The development process is evaluated in parallel, albeit to a smaller extent.

The quality property is extensive enough to be decomposed to the quality attributes that would be investigated with respect to the specific application setting. Additionally, tool support for automatic data collection, metrics computation and reporting will be considered as beneficial. Furthermore, bridging the gap between formal methods research and practical software development, as well as increasing the usability and understandability of formal techniques is an enduring part of the research.

Decomposition of Objective

Critical systems, such as systems for spacecrafts or automotive control components, require treatment that differs in comparison with development of non-critical systems. Since massive money losses or hazardous impact on life and health of people are involved in the case of system failures, these systems have to be dependable. Rigorous approaches and their strict objectives have been applied to software and hardware creation in critical domains [16,87]. They include many specific formal techniques and notations, which in numerous cases are supported by a tool.

In our work, we use measurements to explore the influence of rigorous developments on software systems and identify the regularities observed. We decompose the quality as a system property and investigate a subset of its attributes, which are important from the perspective of dependability property. There are many usability and maintainability, in particular complexity, issues raised when talking about formal approaches. Therefore, we focus on various

facets of the complexity characteristic, which we believe to be crucial when talking about quality control.

We are especially interested in managing and, possibly, mastering the complexity at the system level from the beginning of the developments. We establish metrics specific to given development settings and apply them to case studies provided to us by the ongoing projects. By assigning data from case studies to the created formulas, we enable the computations, which are the foundation for the further analysis. The collected evidence demonstrates the role and impact of investigated formal techniques on systems development.

This thesis presents the solutions, metrics and measurements addressing the following development settings, which are relevant with respect to their adoption in industry:

- Event-B formal method and modelling language, with tool support of Rodin platform
- Statecharts notation with special focus on its distinctive case, Progress Diagrams
- Contract-Based Design applied in Simulink environment.

4.2 Problem Specification and Research Challenges

In this Section we specify the generic problems that we address in this thesis. These are rather broad and each one could separately well serve as an interesting topic for individual research. Therefore, we also limit the scope of the research challenges by defining the success criteria (Section 4.3).

Problem 1: Usability and user-friendliness of formal methods

Formal methods are perceived as difficult to comprehend, strenuous to integrate with the existing business strategy and intricate to combine with the existing tool chain in the developments. Although they are nowadays considerably supported by computer-based tools, there is still a gap between formal methods research and its application in practical software development. This gap needs to be filled or at least reduced in order to facilitate communication and enable finding a shared view on a development. Applied formal methods aim to guarantee correctness of the system and, as a result, could significantly add value to system quality. However, they still need to gain more acceptance outside the formal methods community. There should be a possibility of utilising graphical front-ends in the formal developments, since visualisation increases development awareness. Additionally, general understandability should be amplified, giving a higher-level control over the development. Moreover, a common ground for interaction between academia and industry

should be created to facilitate the knowledge exchange and assists in the development.

Problem 2: Inadequate or not well-documented impact of formal methods on quality of developments.

There is a deficiency of demonstrative data that prove the influence of formal methods on developments. It is one of the main accusations and complaints of people sceptic about the benefits of formal methods. There is a lack of measurement program or suggestions about set of metrics that could be useable in resolving these matters. Only simple and direct measurements have been collected and presented by means of case studies. The case studies used were questioned as being too simple to give scientifically significant results. Moreover, the success stories presented in publications were only scarcely supported by the measurements, if at all. Since formal methods are applied already at the initial development stage, the techniques for quality measurements in this phase are immature and need to be further investigated.

Problem 3: Continuous growth of system complexity as a threat to dependability.

Since software systems are present in everyday life, the list of demands towards them grows. Certain functionality of the system needs to be achieved in order to fulfil these requirements. Therefore, the constant increase of size and system complexity is a natural result of systems getting more sophisticated and feature-rich. There is a need for the techniques that enable the complexity management by detecting the problems of excessive or undesired complexity as soon as they arise.

Problem 4: Insufficiency of software-focused measurements in hardware-oriented system development.

The reduction of manufacturing costs, energy efficiency and advances of the control algorithms has nowadays led to more and more software-intensive systems, where software components are embedded in a hardware predominant environment. At the same time, the products are expected to be of high quality and simultaneously fulfil the growing requirements of the market. These systems, like control systems or embedded systems, are often of high criticality. There are many performance measurements and simulation measures of such systems. However, the software perspective measurements for rigorous methods in cross-domain developments are almost non-existent. There is a need for

establishing software-oriented metrics for modelling and examination of such systems.

4.3 Success Criteria

In this Section we present criteria that we consider as a successful outcome of our research. The problems presented in this thesis in Section 3.2 are further decomposed to sub-problems and tackled progressively. Here we describe them with respect to the limited scope of the problems. In Section *Overview of Research Papers* we indicate the criteria that are completely fulfilled or addressed to some degree.

Criterion 1: Reducing the gap between formal methods research and practical software development

Goal: Increase usability and user-friendliness of formal approaches

Limited scope: Methods in focus of this research problem are Event-B formal modelling language and visual statechart diagrams. The combination of formal and graphical development techniques should support the modelling activity and assist in increasing the usability of the formal development method. Additionally, measurements should provide supplementary feedback for the development teams and managers.

Addresses: Problem 1 and partially Problem 2

Criterion 2: Creating a collection of metrics for measurement and evaluation of Event-B developments

Goal: Enable evidence collection

Limited scope: The goal is to establish metrics and measurements that are specific for Event-B developments. Measurements should also address the problem of control and management of syntactical or data-flow complexity. The analysis of the qualitative and quantitative data should supply direct evidence that supports the decision about adoption of formal methods.

Addresses: Problems 2 and 3.

Criterion 3: Creating metrics for the assessment of formal systems development supported by the use of refinement patterns and measurements when using statechart diagrams

Goal: Provide mechanisms for development control

Limited scope: The aim is to create metrics and measurements explicitly for the development modelled with statechart diagrams. Control and management of structural and data-flow complexity should be provided by measurements and application of refinement patterns. The sub-goal is the analysis of the qualitative data, which will serve as evidence of impact of modelling strategy (here the use of refinement patterns) on the model under development.

Addresses: Problems 2 and 3.

Criterion 4: Establishing a complexity management with metrics and measurement program for rigorously developed Simulink models

Goal: Enable evidence collection; provide mechanisms for complexity control

Limited scope: Development of metrics and a measurement program that are specific for Simulink models should demonstrate the impact of the rigorous Contract-Based Design methodology on the created model. Furthermore, the control and management of structural and data-flow complexity of the Simulink model should be performed with the use of measurements. The objective is to be able to analyse the quality aspects of Simulink models development during the system design process.

Addresses: Problems 3 and 4, also partially Problem 2.

4.4 Research Process

There are a number of publications that have proposed software improvement solutions, e.g. new methodologies, development techniques and tools, without a pragmatic assessment. Hence, the “hands-on” investigations to analyse the rigorous developments for the purpose of evaluating the impact of methodologies on the quality of systems are needed. This is done with respect to certain system qualities (e.g. maintainability and usability) from the point of view of the researcher. Therefore, we emphasise the practicality of the research results.

Research Methods

The goal of our investigation is to provide methods for qualitative studies and experimentation. This will later enable evidence-based research, as shown in Figure 11.

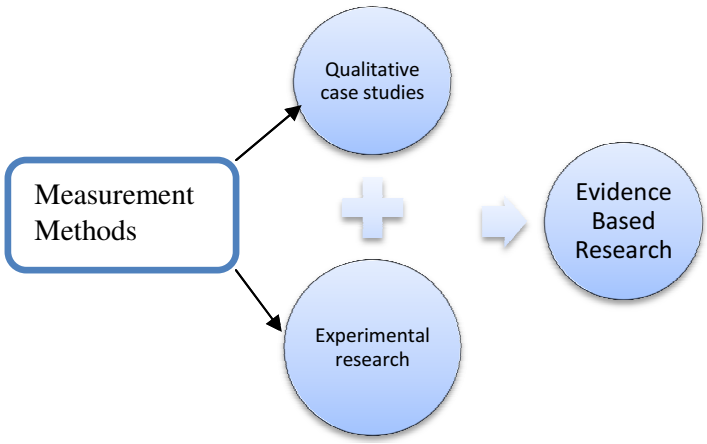


Figure 11. Extended structure of Evidence Based Research

We apply research approaches such as *Design Research* [64] (DR) and *Empirical Research* (ER) [82] in the field of software engineering, rather than business and information systems. We apply Design Research (also called Design Science) which is a constructive and pragmatic approach that aims at establishing useful and effective artefacts. The DR process is shown in Figure 12 [80], where the process steps and the outputs are given. This process is internally iterative, since *Suggestion, Development* and *Evaluation* are often repeated in the DR cycle. An external iteration can also occur and is called *operation and goal knowledge*. There, the artefact is created at the point of *Conclusions* phase and can be used as a *Proposal* for another DR process.

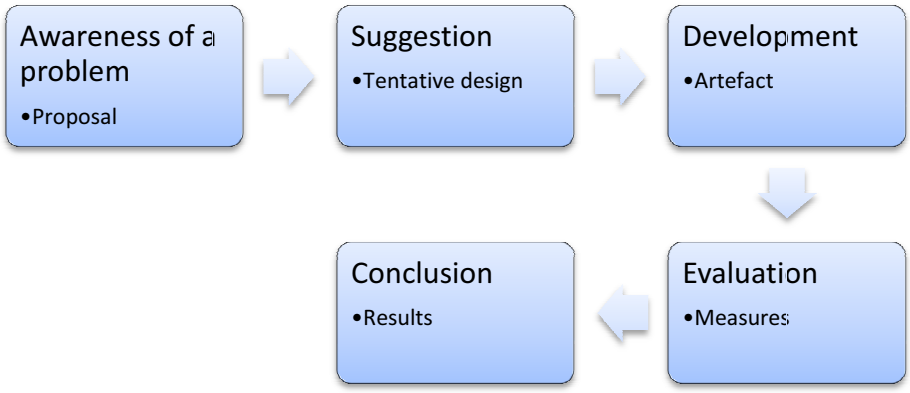


Figure 12. The general methodology of design research.

There are criteria [32] that give guidelines for conducting and evaluating DR. First, DR must generate a feasible artefact, i.e. a construct, a model, a method, or an instantiation (*Design as an artefact).* The main purpose of a design research is to develop and deliver technology-based solutions to significant problems (*Problem Relevance*). A successful DR must supply comprehensible and self-evident help in the areas of the design artefact, foundations, and/or methodologies (*Research Contributions*). Investigating an artefact entails using existing mechanisms to meet the goal, at the same time recognising the specificity of the setting (*Design as a search process*). DR must be presented to technology and management-oriented groups (*Communication of Research*). We have accomplished these criteria in our research process.

However, there are two principles of DR that are not fully satisfied in our work. First one, *Design Evaluation*, which rigorously confirms the utility, quality, and efficiency of an artefact via well-implemented evaluation procedures, is only partially satisfied. The usefulness and quality of the established approaches have been investigated on available case studies and discussed with domain experts involved in projects. Thus, there is a lack of statistical significance. The second principle, *Research Rigour*, denotes that rigorous methods should be applied in construction and evaluation of the DR artefacts. In order to have this criterion fulfilled formal experiments need to be performed.

Design Research has structured our work by giving a skeleton for planning the research. It has given a logical flow and certain degree of control to the investigation and its progress. Since the *Suggestion*, *Development* and *Evaluation* phases are iterative, they provided inner-improvement mechanisms to the research process. The evaluation step required collection of evidence of the practicality and appropriateness of the artefact. The validation was done against knowledge and experience of professionals, as well as real-life and research case studies. Our research is mostly based on the extension of and deriving from the existing artefacts. We foster the existing artefacts by adapting them to rigorous approaches and development setting.

We benefit from Empirical Research (ER) mostly in the development and evaluation stage of the Design Research process. ER it is based on experimentation and observation, i.e. evidence confirming or refuting the research objective. It is applied to solve a specific problem, answer a question or to test a hypothesis.

This approach aims at identification, investigation, authentication and progress of theoretical concepts. We combine research and practice by setting our work in a real world environment due to needs of (industrial) partners cooperating within projects. Finding answers to the emerging problems intends

to establish relevant theory that helps in comprehension of the problem. It also advances the knowledge regarding the problem context. Collecting evidence fosters the existing knowledge and adds to the existing theory.

We believe that combining DR and ER is a successful method of investigation, especially when it is set in the cross-domain context and in the environment where the theory meets practice. The important factor in this work was that the new artefacts are being constructed on what is already known to work and at the same time acknowledge the differences in the research context.

Investigation Techniques

We benefit from the *qualitative* approach, which produces a variety of comprehensive data about (typically) a small number of cases. We chose this approach, since it increases understanding of cases and problem studies, as well as provides a big picture on the Software Engineering research process. We are aware about the reduced generalisability of this research method (to a larger population) and that there often is no definite articulation of problems or solutions (true / false).

Therefore, we complement the qualitative approach with *quantitative* methods. We gather quantitative data that is measurable and use them for the metrics that we established. We use these methods to give exact and analysable expression to qualitative information. Vice versa, we support quantitative techniques with qualitative aspects in order to understand the meaning of the resulting figures. The data collection in our work is direct and, in many situations, automated. It has been done in parallel with the development (on-line context) and by archival analysis of the presented artefacts such as models of the system, documentation (off-line context); both have been supported by observation and analysis. We have interacted with the system developers or experts within the domain to authenticate our findings. These semi-structured interviews, as well as informal discussions and conversations are other data collection techniques that have been used.

In our research, we do the *research-in-the-typical* [83], i.e. we use *case studies* [160] provided by the ongoing projects. We perform in-depth study of rigorous developments to gain deeper understanding of the given problems and to obtain practically relevant research findings. Moreover, we find this type of investigation suitable for doctoral students who cooperate or interact with industry. We used different types of case studies: pre-post (before and after introducing the Contract-Based Design approach to Simulink), snapshot (current view of the Simulink development; statecharts and Event-B development); longitudinal (Event-B development from space domain – observation over time),

each of them supplying different evidence. An alternative to case studies are formal experiments. However, they are not used in our investigations since we cannot provide high degree of control over study. Moreover, we cannot replicate the study due to high costs and risk of experimentation in the domain of critical systems.

There is an industrial context in our work, due to the real life requirements provided by projects we were involved in. The industry partners, who were a part of these projects, stated problems, proposed pilot developments, offered test cases and studies. This brings suitable level of complexity to the research and its relevance and authentication.

Research Structure

The research flow pursued in this thesis is presented in Figure 13 and can be summarised in the following manner. The formal and semi-formal development approaches are the methodologies, the impact of which we have been investigating. The objective is to establish methods for assessment for the formal and semi-formal modelling, as well as provide evidence of the impact of such approaches on the quality of the developments in the early stages. There are two main development settings considered: Event-B method and Simulink.

We explore Event-B modelling setting from the diagrammatic and syntactical viewpoints. For the former type of investigation we establish and explore a special case of statechart diagrams, which also involves application of generic refinement patterns. Moreover, we deal with the maintainability and usability attributes by studying complexity with respect to hierarchical layering of the systems. Furthermore, we investigate the Event-B approach from the maintainability point of view. Establishing syntactical metrics allows us to measure the size and complexity of Event-B specification. In a second track of our work we scrutinise the Simulink models of the system with respect to the maintainability attribute. We examine complexity of a model and model interdependencies.

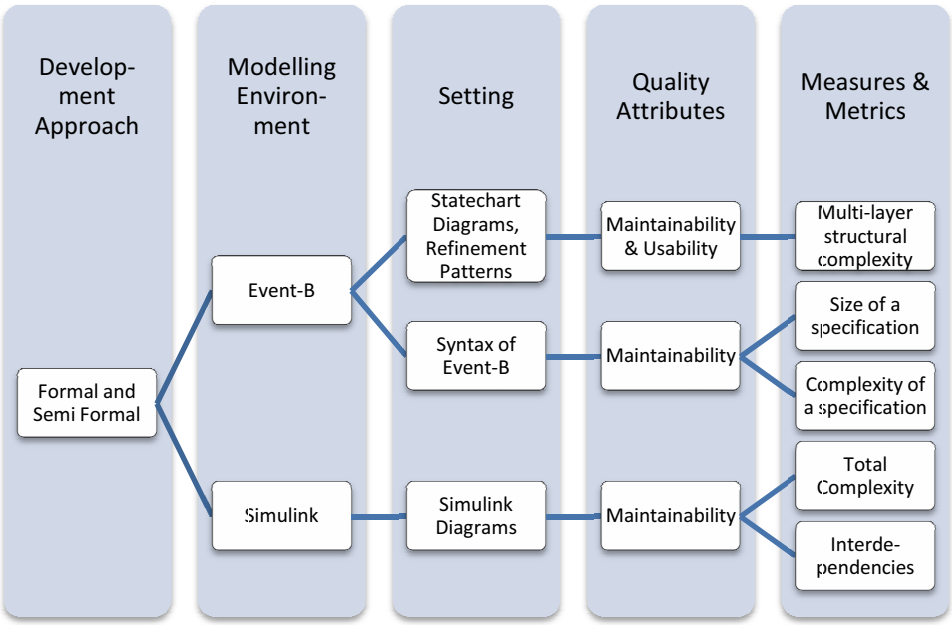


Figure 13. Structuring of research

We believe that empirical studies are needed in software engineering, e.g. to evaluate development process activities, as well as their impact on the quality of the development. They involve also the human aspect, which should be mentioned in the analysis of the validity of investigation. However, in our work the human aspect is considered solely when discussing the results of our work with domain experts. The evidence recording activity, which consists of data collection and their analysis with respect to the goal or research objectives, is followed by reduction of the data set (outliers). It captures and demonstrates the issues that were functioning only as a tacit knowledge. The interpretation of research results that is based on engineering principles is the cornerstone for decision making and advances the engineering discipline.

There is also the validity issue that needs to be mentioned when talking about the research output and performing empirical studies. There is a lack of validated results in the field, which is mainly caused by the problems with study replication or complete deficiency of such studies. Validity to some degree can be achieved by case studies and in this form it is accomplished in our work. We also support our validation process with observation. This is the case when talking about metrics research, where even relationships between different measurements can be thought of as a metric.

5 Overview of Research Papers

The most significant results of the work presented in this thesis are documented in six papers given in Part II. In Figure 14 we illustrate the relations between the papers, where lines with arrows indicate the direct research flow, whereas the dashed lines show the connection between the results with respect to the measurements topics.

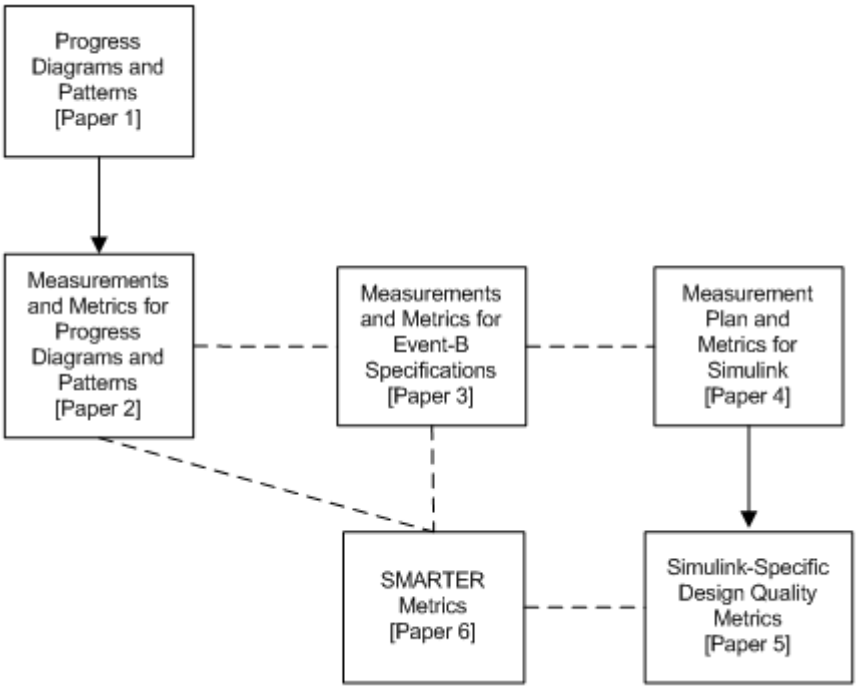


Figure 14. Relations between the papers presented in this thesis

In this chapter an overview of the research publications shown in Figure 14 is provided. First, the involvement of the author for each paper is indicated. Then the overview and contribution of each paper is presented. Moreover, the research process is described and the relations between the publications are depicted.

Paper 1

M. Pląska, M. Waldén and C. Snook, *Documenting the Progress of the System Development.* In The Book on Methods, Models and Tools for Fault Tolerance. A. Romanovsky, M. Butler, C. Jones, and E. Troubitsyna, (Eds.), LNCS 5454, Springer-Verlag, Heidelberg, 2009.

Author's contribution: Providing a case study and application of the design patterns. Co-creator of Progress Diagrams. Partial responsibility for the publication.

Main topics: In this paper we introduce the idea of Progress Diagrams, which represent snapshots of the current refinement step in the system development. Progress Diagrams are tightly connected with the Event-B formalism. They are intended to support rigorous developments by providing graphical assistance throughout the modelling activities. Furthermore, they document design decisions and illustrate the application of certain refinement patterns. Moreover, they indicate how the problems in the system design can be detected more easily and give intuition of the correctness of each refinement step with regard to refinement rules (Proof Obligations). The use of Progress Diagrams is illustrated by a case study.

The research process for this publication initiates from noticing the need of facilitating the uptake of formal methods. This can be achieved by the use of visual constructs, which are more intuitive than syntactical elements. User-friendliness and usability of formal approaches, here Event-B [3], was already facilitated by a graphical tool integrated with Rodin platform [121], i.e. UML-B tool [144]. However, an approach being more focused on the intensively developed part of the system could be of benefit. The objective is to combine development rigour and reasoning about the model properties with ease of use, as well as concentrate specifically on certain parts of the system. The tabular form that combines elements from Event-B syntax and a visual representation of a part of a system in a form of a statechart diagram is the artefact developed in this research. We call it a Progress Diagram, since it depicts the progress of the system under development.

The evaluation was done against a case study, which showed the development of a memory scheduler system. We enabled reasoning about the parts of a model under construction, as well as provided a perceptive and compact view on the model. The established approach provides additional control over the development and is useful for documentation purposes. In the future it can be extended with tool support via one of the Rodin platform plug-ins. Moreover, additional mechanisms for the management of development progress, e.g. with measurements and metrics, would be beneficial.

This publication addresses Problems 1 and 2, i.e. the issue of increasing usability and user-friendliness of formal methods by the use of patterns and visual techniques. Moreover, maintainability of a specification is one of the objectives, since analysability and understandability of a formal model is facilitated. Furthermore, documenting impact of rigorous methods on the developments is considered.

Paper 2

Marta Olszewska (Pląska) and Marina Waldén. *Measuring the Progress of a System Development.* In The Book on *Dependability and Computer Engineering: Concepts for Software-Intensive Systems.* L. Petre, K. Sere, and E. Troubitsyna (Eds.), IGI Global, July 2011.

Author's contribution: Initiator for the work in this paper and the main author. Main responsibility for the publication.

Main topics: This paper is a continuation of research on Progress Diagrams and refinement patterns defined in Paper 1. Some direct measurements for Progress Diagrams are identified, as well as indirect structural measurements for the diagram are created with regard to the progress of a system development. Multi-layer structural complexity metric is established and analysis of the complexity characteristic is performed with respect to refinement patterns. We also tackle orthogonal and hierarchical decompositions and their impact on the complexity issue.

The research for this publication is a continuation of the one presented in Paper 1. Our conjecture is that metrics and measurements can assist the development at its early stage. Therefore, we focus on metrics, measurements and complexity control from the perspective of Progress Diagrams. The objective is to create a measurement program for the visual part of Progress Diagrams. Therefore, we identify direct measurements, which can be obtained explicitly from the model. Moreover, we establish structural complexity of the model, which is rooted in the McCabe complexity [99]. We extend the metric to take into consideration the hierarchical structure of the design of the system, as well as constructs like orthogonal states. The assessment and validation of the multi-layer structural complexity metric is first done with small examples. Then, in order to give an intuition behind the complexity metric, we illustrate the application of the metric with a test case consisting of several refinement patterns given in Paper 1.

The evaluation of the complete approach was also presented as part of Paper 2. We concluded that our approach depicts the impact of the design decisions on the complexity of the model under development. Furthermore, it can be generalised to statechart diagrams and provides evidence of the impact of methodology on the development, i.e. design. However, for the method to be scalable and practical, the measurements activity needs to be automated. Moreover, further validation of the metric needs to be done against numerous large case studies in order for the metric to gain statistical significance.

The work tackles Problems 2 and 3. In particular, it provides metrics as means for collecting measurements. These are necessary for documenting the

impact of rigorous methods on quality of developments. Moreover, measurements and patterns are the instruments used for the management and control of the complexity growth. The maintainability sub-attributes, in particular analysability, manageability and understandability are in the scope.

Paper 3

Marta Olszewska (Pląska) and Kaisa Sere. *Specification Metrics for Event-B Developments.* In Proceedings of the CONQUEST 2010, 13th International Conference on Quality Engineering in Software Technology, Dresden, Germany, September 2010.

Author's contribution: Originator of the work for this paper and the main author. Main responsibility for the publication.

Main topics: In this paper the focus is on establishing the metrics specific for Event-B modelling setting. The main goal is to be able to asses the specification from the quantitative point of view. The syntax-based metrics for the Event-B language regarding dynamic and static parts of a specification are established. Also statistics about proof obligations are used, as the proving activity is considered as a vital element in the process of creating a specification. Obtained results, which regard size, difficulty (complexity indicator) and effort of constructing a specification, are analysed from the perspective of an abstract specification and its consecutive refinements.

Research on metrics for Event-B developments is initiated to gain a measurement viewpoint on formal modelling. Previously we focused on the rigorous developments supported by the visual artefacts and concentrated on metrics for the graphical representation of the system. We have also been encouraged by the ongoing project, which indicated the need for evidence of the impact of formal methods on the early stage developments. Again, we work towards assessment of the complexity aspect of the design that is created during formal modelling of the system. In order to tackle complexity from a different viewpoint than the one of flow-graph, we propose metrics dedicated to the specification language and based on its syntax. We adapt Halstead Software Science [58] by choosing the primitives and defining the meaning of metrics specifically for the Event-B setting. In the next research iteration, we incorporate the refinement mechanism into the metrics.

The approach has been applied to several case studies, i.e. a large and complex industrial case study from the space domain, as well as a number of smaller case studies. The data collection and computation of the metrics has been performed automatically. The validation of the approach was done by observing and confirming the relations between the metrics. Although our results

from these case studies confirm the appropriateness of metrics, we believe that they still can be modified to be more accurate, e.g. by adding weights to certain primitives. Moreover, more industrial case studies are needed in order to fully validate our findings.

This paper addresses Problems 2 and 3, since it provides mechanisms for documenting the impact of formal modelling on the size and complexity of the specification. Moreover, the presented metrics enable modellers with a constant control over the complexity. As such, they positively influence the maintainability attribute and thus the dependability aspect of the development.

Paper 4

Marta Olszewska (Pląska), Mikko Huova, Marina Waldén, Kaisa Sere, Matti Linjama. *Quality Analysis of Simulink Models*. In Proceedings of the CONQUEST 2009, 12th International Conference on Quality Engineering in Software Technology, Nuremberg, Germany, September 2009. dpunkt.verlag GmbH Heidelberg, Germany.

Author's contribution: The main author, initiator of the idea of establishing the measurement program for Simulink developments. Test case from the domain of digital hydraulics and its presentation in the paper were provided by Mikko Huova and Matti Linjama (Tampere University of Technology, Department of Intelligent Hydraulics and Automation).

Main topics: This paper tackles the problem of assessment of Simulink-specific developments from the perspective of software quality. The established measurement plan considers the development process and product (Simulink model) measurements. A Simulink specific complexity model is created, which includes structural and data complexities. Complexity analysis is performed with respect to the hierarchical structure of Simulink model. Quality investigation assesses the impact of Contract-Based Design methodology used in the hydraulic controller development.

This work originates from the observation that the hardware and software domains are tightly connected and there is increasingly more software elements built in hardware. The high-level objective of our work is to facilitate and boost the development of hardware areas, here digital hydraulics, through advances that originate from software aspects. Our purpose is to create quality measurement plan, which will include direct and indirect measurements, and conform to the perception and experience of the developers. The work involves product measurements, i.e. direct model metrics, as well as defect related measurements. Moreover, the development process is analysed from the perspective of the impact that the semi-formal use of Contract-Based Design

[19] methodology has on its quality. We establish a complexity model based on the Card and Glass complexity [29] and foster the metric by adapting it to the Simulink [132] development setting. We define the meaning of the model elements with respect to the metric constructs and characterise relations between them. Since Simulink allows for the system to be modelled hierarchically, our metrics are created to enable us to assess the model on the level of a layer. The entire process of metric creation and establishing the measurement plan is done in small iterations.

The evaluation of the approach for this publication was done against a large and rather complex case study from the domain of digital hydraulics. We consulted our findings with the developers from the digital hydraulics domain, who confirmed the meaningfulness of the results. It should be mentioned that our approach was also applied to another large case study [70]. To a smaller extent, due to retrospective character of investigation and limited availability of data, it can also be found in [119]. At this point, our approach needed automation, to facilitate the data collection and analysis. Our observation is that the best development practices can be singled out with the support of metrics, measurements and analysis and serve as guidelines to less experienced developers, e.g. building control systems.

Mainly Problems 3 and 4 are **tackled in this publication** by establishing software-oriented complexity metrics specific for the Simulink development environment for the purpose of the control and improvement of the developments. We focus on maintainability attribute, in particular analysability, manageability, modifiability and understandability sub-attributes. Problem 2 is partially addressed by using the metrics to assess and give evidence of how rigorous development method impacts the design.

Paper 5

Marta Olszewska (Pląska). *Simulink-Specific Design Quality Metrics.* TUCS Technical Report number 1002 Turku (Finland), March 2011

Author's contribution: the sole author.

Main topics: This paper is a continuation of the research on metrics for Simulink-specific developments described in Paper 4. We further explore the complexity characteristic by identifying the outliers. Instability and abstractness metrics for Simulink are defined and based on structural properties of the models. The relation between these two is a basis for identification of potential problems in design interdependencies. All metrics are analysed from the perspective of possible design issues. The main objective of this investigation is

to incorporate measurements to the development process, and thus support the modelling activity within Simulink environment.

This research process can be interpreted as a large loop iteration for the work presented in Paper 4. We proceed with the research on metrics and extend the complexity metric with the concept of identification of outliers. This aids in finding the possible fragilities in the design by indicating the too complex or too simple parts of the system. Moreover, we investigate the structural complexity issue with respect to design interrelations. We transfer the concept of inter-dependency from object-orientation [91] to the Simulink [132] environment. We define the model elements and relations between them in terms of internal dependencies. We also analyse the meaning of connections between the elements of the design. The purpose is to support the developers during the modelling of the system and raise their awareness about the consequences of the design decisions.

The evaluation of the approach was made against the same case study we used for Paper 4. The data collection, as well as the computation of metrics was done automatically [20]. The presented approach is scalable due to the tool support and can assist the developers upon request also during the modelling of the system. We believe that there is a need for further validation of the presented metrics against large and, possibly, industrial case studies in order to confirm their meaningfulness.

The publication advances the work carried out within Problems 3 and 4, and to a small degree tackles Problem 2, since it deals with the complexity issues in the hardware setting. The approach presented in Paper 4 is extended in this paper and directed towards measurements being regarded as the quality indicators, which provide guidelines during the development. We concentrate on the maintainability sub-attributes, in particular the analysability, stability, manageability, modifiability and understandability sub-attributes.

Paper 6

Marta Olszewska (Pląska). *SMARTER Metrics.* Accepted to the 5th World Congress on Software Quality, October 2011, Shanghai, China.

Author's contribution: the sole author.

Main topics: This publication abstracts from the current work and describes the metrics and measurements from a higher-level perspective. It describes our vision on the quality metrics and measurements and their current position in the development of software systems. It proposes the idea of SMARTER Metrics, which share desired key characteristics. The definition and our insight on these

characteristics is the central part of the paper. The presented concept is supported by an example of a set of metrics that have been established in our previous work.

We transfer the generic ideas used in the evaluation and improvement of the business success (SMART criteria) to the software setting. We identify and extend these features from the perspective of software quality metrics. We believe that satisfying these should be a conceptual foundation of a feasible metric. SMARTER denotes specific, measurable, actionable, relevant and timely, but also easy to manage and reusable. Metrics that have these characteristics enable the dynamic tracking of the development progress by targeting significant artefacts. They should be tool supported and, if possible, transferable between the projects, teams and application domains. Since the established framework is only a proposal of metrics having certain preferred features, it cannot yet be validated. However, it is based on our previous work and gives good foundations for further investigation.

This publication presents the common characteristics of all the metrics we developed. It is essentially an abstract view on the work presented in the publications related to measurements and metrics. **It addresses** Problems 2, 3, and 4 from a higher-level viewpoint. All possible quality attributes are included in this work and tackled in a generic manner. Problem 1 can be linked to this publication to a small extent.

Overview

There are many commonalities in the research problems tackled in our publications, just to mention the assessment of rigorous developments, collection of measurements, establishing complexity metrics and providing quality control. The diversity of the types of development approaches and settings, the kind of measurements that are collected and types of complexity is also noticeable. Nevertheless, quality as a desired feature remains as a cornerstone of our research. A high-level view on our work and the problems that we addressed in this thesis is demonstrated in Table 1.

Success Criterion	Paper 1	Paper 2	Paper 3	Paper 4	Paper 5	Paper 6
1. Reducing the gap between formal methods research and practical software development	+++	++	+	+	+	+
2. Creating a collection of metrics for measurement and evaluation of Event-B developments	+	+++	+++	-	-	++
3. Creating metrics for the assessment of formal systems development supported by the use of patterns and measurements when using statechart diagrams	++	+++	+	-	-	++
4. Establishing a complexity management with metrics and measurement program for rigorously developed Simulink models	-	-	-	+++	+++	++

Table 1. Cross-listing of Papers realising the Success Criteria

In Table 1 we show the overview of the Success Criteria that we defined earlier in this thesis, and juxtapose these with the research performed in particular Papers. We also indicate the degree of coverage and applicability of our results to the Success Criteria with a plus symbol '+' on a three level scale: '+++', '++' and '+', denoting large, medium and fair, respectively. The symbol '-' signifies that the material included in the Paper does not concern certain Success Criterion.

6 Achievements - The Overall Picture

The contribution of this thesis spans over a number of application settings, metrics and measurement attributes. However, all the results of the work converge to certain levels of quality of critical software systems and related artefacts from the perspective of rigorous developments. In order to be able to control the development of the system it is crucial to have metrics and measurements that support it. Formal developments are not well researched with respect to these. We contribute to software quality management, herein the dependability property, by creation of measurement plans and by establishing metrics for the artefacts in the initial stages of the development. The focus is on the maintainability of a model and a specification, as well as usability of formal approaches.

We find it beneficial to use measurements to indicate the potential development problems and give guidelines towards the possible remedies. Additionally, we concentrate on the issue of leveraging the complexity of the system early in the development, by e.g. using patterns and quality measurements. According to our observations, structuring and controlling the development during modelling of a system results in its higher maintainability and understandability already at the design stage.

We demonstrate that there is a need for multi-domain cooperation, where specialists from various backgrounds share their knowledge for the purpose of obtaining good quality, efficient and correct system. Based on the case study from digital hydraulics area, we show that professionals building control systems for hardware and experts from formal methods field benefit from the cooperation. Moreover, we have evidence that this collaboration is a success. Additionally, we confirm that the software-oriented techniques for the assessment of quality in the early stages of the development are indeed needed for the evaluation and future process improvement.

In our work, we address the usability, understandability and maintainability aspects of modelling with statecharts. We combine the rigour of Event-B language and UML-like visualisation by creating Progress Diagrams. Progress Diagrams allow developers and managers to concentrate on the part of the system being intensively developed or to focus on the part of the design that is of the biggest interest. Furthermore, they provide a compact picture on the development flow and design decisions, as well as an intuition on refinement steps and the needed proofs. This indirectly increases the usability and understandability of the Event-B modelling by giving insight into formal designs

with the use of graphical notation. Moreover, Progress Diagrams help to manage system complexity due to clear and comprehensible documentation.

We also establish a multi-layer structural complexity metric, which is derived from the McCabe complexity. We apply it to the top-down refinement-driven development for Progress Diagrams. This metric facilitates the control of complexity over the modelling of the system by enabling quality measurements and quality assurance activities from the perspective of design maintainability. It additionally provides us with the quality measurement aspect of modelling by linking formal and precise Event-B and ambiguous UML modelling. It is worthwhile to mention that the multi-layer structural complexity metric is extendable for statechart diagrams.

Additionally, we contribute to assessment of quality, in particular maintainability, for the Simulink modelling environment. We establish software-oriented metrics and provide a measurement plan that would facilitate evaluation of Simulink models during the conceptual phases of the development. In particular this is applied to the specification and design phases. A complexity model consisting of structural and data complexity is established and a method of identifying the outliers is defined. This allows us not only to observe the complexity characteristics of different levels of the Simulink model, but also identify the possible fragilities that can lead to problems or defects later on. We demonstrate that abstractness and instability metrics known from object-oriented design also function for Simulink models and serve as indicators of the overall quality and manageability of the design.

The aspects that we address in our work also regard creation of metrics for the Event-B specification language. It is essential to establish and apply the measurement methods that support formal design and analysis of the model during the modelling activity. Collected measurements are the additional means that aid tackling the complexity issue, thus maintainability and related sub-attributes. We enable monitoring of the Event-B developments with measurements, which consider the refinement mechanisms and the syntax of the language. This supervision can be useful when modelling, using patterns or decomposition and abstraction mechanisms.

By providing a SMARTER metrics framework we summarise the experience we built up when working on metrics and measurements. We propose an idea of metrics that are characterised by desirable features, meaning that the metrics are specific, measurable, actionable, relevant and timely, but also easy to manage and reusable. The framework is a conceptual foundation of a repository of metrics, where each metric is suitable for the assessment of certain characteristic, but at the same time abstract enough to be adjusted or extended to a different development setting. In this work, all the quality attributes are involved.

In Table 2 we summarise the results of our work, by providing an overview of the artefacts developed in our research. We present the generic and initial artefacts, which we regard as a necessary background and the starting point for our investigation. Then, we give the outcome of our research with respect to the development setting. We also present the related research areas and artefacts, as well as quality attributes that are in focus. We benefit from utilising the underlying knowledge and existing theories by using them as building blocks for creation of new artefacts. These capture the most practical and feasible tactics in accordance to the development setting and features specific for certain formalisms.

	Initial artefact	**Our result**	**Setting**	**Related to**	**Quality attributes**
1.	Statechart Diagrams	Progress Diagrams	Statechart Diagrams and Event-B	UML, Event-B, UML-B	Usability & maintainability
2.	McCabe Complexity	Multi-layer Structural Complexity	Progress Diagrams	Statechart Diagrams, Event-B, Model / Structural metrics	Maintainability
3.	Halstead Software Science	Specification Metrics for Event-B developments	Event-B specification	Syntactical metrics	Maintainability
4.	Card and Glass complexity	Complexity for Simulink models	Simulink model	Model / Design metrics	Maintainability
5.	OO package metrics by R.C. Martin	Interdependencies metrics for Simulink models	Simulink model	Model / Design metrics	Maintainability
6.	SMART project success criteria	SMARTER Metrics	Software systems (general)	Software metrics (general)	All (generic view)

Table 2. Initial artefacts and the outcome of our research with respect to development environment.

Naturally, the work included in this manuscript will not suffice as hard evidence for the decision making on the take-up of formal approaches. However, we believe that it gives outcome that can be used for the discussions regarding the transfer of formal methods to organisations. We have shown it is a good foundation for further research in this area and an indication of the need of cross-domain cooperation.

7 Related Work and Dedicated Literature

Achieving dependability is a crucial factor especially when talking about quality of critical computer-based systems. For many years now this intricate task is supported by the application of rigorous approaches. The suitability of these development techniques needs, however, to be evaluated in order to assess the actual impact of the methodology on the quality of the final system.

In this chapter we present related work according to several research perspectives, i.e. the dependability aspects of modelling, the role of measurements for rigorous developments, controlling the quality of developments, as well as the quality concept in Simulink environment.

7.1 Dependability Aspect of Modelling

Meyer presented many ideas about dependability and techniques for achieving it in the form of a survey [104]. One of the described approaches is design by contract, which can be used at the specification level and help developers with more precise design. We believe that our Progress Diagrams, that support the design process with measurements, present the system in a more thorough and understandable way. Moreover, the diagrams provide better documentation mechanisms and can be easily used by developers and managers to comprehend the system at certain levels of abstraction.

According to Jackson [77], design is regarded as means of achieving dependability. The main reasoning is that the poor design of software, which is nowadays present in everyday lives, is the main cause of the failures, and thus high costs. The proposed solution towards good quality software is the evaluation of the design, as well as simulation of every state that the software can take in order to verify that none leads to a failure. The tool support Alloy, promoted by the article, has been proved useful for obtaining precise, robust and thoroughly exercised designs. It uses checks to find conceptual and structural design flaws in software. In our work, we assess the design with measurements in order to control and manage it from the complexity perspective.

As stated by Jackson in one of his journal articles [76], dependability is being decomposed with respect to its meaning not only to software, but also in its role in society. It is presented as a trade off between benefits and risks with some level of assurance. One of the topics that are discussed on an abstract level is complexity management for the purpose of achieving a system that is as simple as possible. Although this task is not cheap and easy, it is indicated as effective, especially in the design stages. The comparison is made with the costs of failures

and excessive complexity of the final system. A rigorous and credible development process is regarded to be one of the key factors of high quality software. We give a specific and practical solution to the general concept presented by Jackson. In our work, we propose a concrete methodology for the thorough design, which is supported by a chain of evidence (documentation) and measurements.

A mixed-criticality type of systems is considered by Jackson and Kang [78], where a separation of concerns with respect to the criticality of the issues is proposed. They suggest using decomposition mechanisms and decoupling to achieve a robust design. Jackson and Kang recommend that more focus should be placed on the fundamental problems of design and modelling related methodologies. Our research meets their expectations in the matter of using design and its analysis in order to contribute to the control over the constructed system. In this way we increase the dependability in general, in particular maintainability or resilience. We additionally provide the mechanisms to facilitate tackling the modelling-related issues.

A practical approach for achieving dependability and security (also resilience) that targets the developers is given in [102]. Several key activities and recommendations, which are required for integrating security and other non-functional requirements into Software Development Lifecycle (SDLC), contain the system design phase. In our work, we focus on the design by concentrating more on modelling the general behaviour of the system, than on integrating a security and resilience mindset in the development process.

7.2 Measurements for Rigorous Developments

The impact of design flaws is discussed in [36], where the authors investigate and confirm the relationship between software defects and a number of design flaws in several open source systems. They relate code defects and trace them back to the design. We, on the other hand, focus on evaluating critical systems throughout their development. We observe the defects correlation with the design phase, having in mind the rigour of the development method in, e.g. the Simulink setting. We talk about possible design fragilities, which later may lead to defects.

In our research considering Event-B or statechart diagrams, we do not discuss defects *per se*, since we remain at the specification phase and code is not generated from these specifications. A different approach is proposed in [117], where defects from source code are traced back to the weak points in the design. The authors propose the assessment of quality of the design, which is based on UML models that are retrieved from the source code. Reverse engineering

technique is thus used as an input for system design evaluation. The authors give insight in system design and existing interrelations through the set of metrics on a system and class level. Metrics have also been used for assessment of the quality of model transformations [146,147] for certain model transformation languages. They assess e.g. the complexity of a model transformation regardless of the type of the created system. In our work, we investigate the impact of complexity characteristic specifically on the design of the critical system.

There are several publications or reports addressing certain critical systems or rigorous developments [87,25,118,158,88]. They consider mostly straightforward measurements; some applied to well-known examples, such as the case of the driverless Metro line number 14 in Paris, others to formal specification languages, such as Z. An assessment and certification of the safety critical software system (the above mentioned driverless Metro) was done by El Koursi et al. [39] Interesting results are presented in an extensive survey on industrial adoption of formal methods within the last 20 years [157]. It describes a chain of industrial projects, together with some remarks originating from the survey and additional experience documentation. It gives a broader view on the role of formal methods in industry. Our work provides more intricate techniques for the assessment of complexity in certain formal development settings; however, our experience reports and evidence based on case studies range over a time span of a few years. Nevertheless, we feel that with our results we contribute to the industrial take-up of formal methods.

Some work on measurements that support the modelling process was done for the Z specification language [63] and for estimations in B-Method [162]. Therefore, having measurement-supported modelling in Event-B appears beneficial and will make this method even more attractive for the industrial setting. A prediction of the erroneous parts of the specifications for the Z language was done in [150]. Due to the lack of historical data we have not established any prediction models, yet. However, in our work we have provided the mechanisms for data collection, thus enabling the gathering of evidence.

7.3 Controlling the Quality of Developments

There have been many publications providing guidelines on how to employ formal methods [22,23] and what caused the misfortunes and prejudice against these approaches [65,57]. Moreover, there have been attempts to incorporate formal specifications in software development [49], as well as creating and validating maturity measurement models for specifications [50,145]. They all emphasized the importance of attaining software quality, particularly in critical systems, with the successful use of formal methods. However, they tackle the

problem from the global level, i.e. they do not indicate the formal method used or the application area of the method. We investigate the impact of specific formal methods on the developments that are set in certain environments. We go deeper into the problem, since we believe that the lower-level approach provides us with more thorough information. The take-up of formal methods by organisations is supported with the measurement mechanisms for collecting evidence. These are specific for the method and setting, and can provide modelling guidelines and act as demonstrators for successful software developments.

The modelling rules are a part of the "Code Complete" book [101], which is a guide consisting of the most effective techniques and must-know principles for software construction. Modelling has also been investigated with respect to Event-B setting by Iliasov et al [72,73,74]. The research concerned the application of refinement patterns, in particular fault tolerance patterns, during the system modelling [72]. A set of component based patterns for developing embedded system designs was proposed and formally verified in [141]. An extended form of state machine is used, which additionally supports reactive and time-triggered behaviours. In our work, we also use refinement patterns for the purpose of improving the structuring of the system; however, the patterns we apply are more generic.

Structuring the formal design and fault tolerance in the system by the use of modes was suggested by Iliasov et al. [73]. Further research involved developing a so called model critic plug-in [71], which gives feedback to the modeller about the quality of the model with regard to some earlier defined criteria, during the construction of the system. One of the results of our research was providing the modellers with quantitative information about the model, i.e. its size and complexity. This data is also available while constructing the system and can be used as an indicator of a bad design, e.g. too complex refinement steps.

The design decisions are often tool-supported, as presented in [61], where the tool aided system analysts with the trade-off study of quality based on the prediction values. The tool was intended for assessment of quality-of-service attributes, i.e. performance, reliability and maintainability. In our work, we indirectly assessed maintainability by observing the static representation of the system, e.g. model or specification. As a separate result of our investigation, we established measurement-based guidelines that can be provided to the developers during the modelling of the systems in the Simulink environment.

7.4 Support for Quality in Simulink

Simulink is quite well supported with tools and techniques, which aid in achieving high quality designs of dependable control systems. There are various modelling guidelines provided by MathWorks Automotive Advisory Board [93], however, some of them were found to be similar to each other (repetitiveness) or (seemingly) contradictive [41]. The toolboxes, on the other hand, are commercial and dedicated to specific tasks. For example Simulink Design Verifier [97] is related to rigorous approaches by assuring that software fully complies with all the expected requirements. It affirms the quality of the developed system by generating tests and proving model properties using formal methods. It computes different types of model coverage metrics based on the model structure, showing the dead branches or unnecessary elements. On the other hand, Simulink Verification and Validation [98] toolbox verifies models and generated code, as well as provides checks for the DO-178B and IEC 61508 industry standards. If the toolbox is present, one can benefit from the Model Advisor [94], which checks a model or part of it for conditions and configuration settings that can lead to imprecise or inefficient simulation of the system that the model represents. Furthermore, there is also a built-in diagnostic command 'sl_diagnostics' [95], which collects direct data about the model. An open-source tool that is a front-end for this command is also available in MathWorks resources [67]. In [68] authors describe the meaning of collected data with respect to the concept of quality and productivity.

Although there is plenty of material considering construction of quality software, it is either hardware or performance oriented, or regards the direct measures of the model. We proposed to support the modelling activity with software-focused measurements, which are based on the metrics founded on the structure of a model. Since the open-source, Eclipse-based and extendable tool support for Simulink has been developed within our laboratory [20], we were able to implement the metrics so that the feedback is automatic and given upon request. This way we complement the existing approaches that aim at high quality, dependable software systems.

8 Discussion and Conclusions

In this chapter, we give a succinct overview on the current state and tendency in the measurement activities in the (semi) formal approaches. We also present the limitations of our research and give potential concepts for future work. We finally concisely summarise our work with some general remarks about our contribution to the area of software system measurement within the software engineering discipline.

Formal methods are an approach that is well-known and appreciated not only in research areas, but also gaining acceptance in industry. Since there is more evidence of the success of rigorous approaches and numerous stories about their application in various domains, they are more understood and recognised in industry. The emphasis remains on the adaptation and application of formal methods and the possibility of technology transfer from research to industry. In return, industry identifies the open challenges and problems that can be tackled by academia. This state of the art knowledge exchange and fostering the position of formal methods has been one of the goals in the European Project DEPLOY (2008-2012) [37]. The outcome of the project will include the qualitative and quantitative evidence of industrial take-up of formal methods. We contributed to this topic with publications related to Event-B, i.e. papers regarding Progress Diagrams and metrics for these, as well as metrics for Event-B language.

The interest of industry in formal methods is confirmed by many applications and practical experience [87]. Formal methods are also recommended or highly recommended practices when licensing critical software, just to mention the IEC 61508 standard ("Functional Safety of Electrical, Electronic and Programmable Electronic Safety-related Systems") [142,1] or ISO 26262 standard for automotive domain ("Road vehicles — Functional safety") [75]. The application of rigorous methods is additionally followed by measures and techniques that are obligatory for the product to be certified. We find it beneficial to investigate metrics and measurements for these development types, since they can also support the qualification process.

8.1 Discussion on Limitations of the Approach

We are aware of the limitations of our techniques and artefacts presented in this thesis. We divided them according to the threats to their *validity* and the presence of *tool support*. We find these characteristics crucial for the practicality and relevance of our research.

Automating the computation of metrics and collection of measurements simplifies and lowers the cost of the quality evaluation process. Moreover, it minimises the risk of human error in gathering direct data and performing calculations on these. Validity of the approach and its evaluation, on the other hand, is necessary for the method to be meaningful and possibly improved or fine-tuned in the future.

Threats to validity are classified into four types: *construct validity*, *conclusion validity*, *internal validity* and *external validity* [156]. Construct validity is identified as the ability to measure the artefact being studied, whereas conclusion validity is the ability to draw conclusions based on statistical inference. Internal validity is characterised as the ability to isolate and identify factors affecting the studied variables without the researchers knowledge, while external validity is the ability to generalise the results. This section is organised according to the types of validity and the sequence of papers presented in Section 4. Since Paper 6 presents a generic proposal for the metrics framework, it is not a subject of our validation study. We describe the results of our validity evaluation by indicating the possible limitations with respect to artefacts. A high level overview of the limitations with respect to the artefacts presented in this thesis is given in Table 3.

	Paper 1	**Paper 2**	**Paper 3**	**Paper 4**	**Paper 5**
Tool Support	No	No	Yes	Yes	Yes
Construct Validity	Yes (no tool support)	Yes (no tool support)	Yes	Yes	Yes
Internal Validity	Yes	Yes	Yes	Yes	Yes
Conclusion Validity	Yes (more experi-mentation needed)	Yes (larger case studies needed)	Yes	Yes	Yes (more evidence needed)
External Validity	Yes. Validated against a case study.	No. Validated against examples	Yes. Validated against a large and complex industrial case study	Yes. Validated against a large and complex case study	Yes. Validated against a large and complex case study

Table 3. Limitations of the developed artefacts

Realising *construct validity* in our research was done by confirming the relationship between theory and observation of the artefact under study. All our research was focused on developing and (or) measuring the artefacts and finding evidence that the artefacts we chose were sufficiently representing what we intended to investigate. For instance, the results of the application of metrics which were created for the Event-B language confirmed the intuition behind the model evaluation. When discussing construct validity of our research there is one problematic issue regarding Progress Diagrams. Due to lack of automation in the data collection (no tool support), the precision of the gathered data is uncertain, since human errors could be brought in early in this assessment process. However, in this particular case, the data were carefully gathered and the results double-checked. Therefore, we can determine that the construct validity is fulfilled to a large extent for all of the created artefacts, including Progress Diagrams.

The variables and factors that could influence the investigation should be taken into account and, if possible, limited when considering *internal validity*. The objectivity of measurements could have been biased due to subjective analysis of the results by the researcher. However, the outcome was discussed and confirmed by the domain experts, i.e. developers, research partners, as well as company representatives. The presence of the researcher alone did not impact, nor bring the reactive bias to the results of a study. Therefore, we reason that the internal validity is appropriately addressed.

We are aware of the risk of oversimplified assumptions, which impacts the soundness of results and is a serious threat to *conclusion validity*. The weakest case in our research is the investigation regarding measurements for Progress Diagrams. It is the only study that has been validated simply against examples and thus needs more experimentation. The work on Progress Diagrams as constructs, on the other hand, has been validated against a real-life case study. Event-B metrics and measurements have been confirmed against a large and complex industrial development from space domain, whereas the research regarding Simulink was supported by a large and complex case study in the digital hydraulics area. Nevertheless, we feel that for the Simulink measurements presented in Paper 5 some more empirical studies would be beneficial. In all presented cases there is a need for more case studies and experimentation regarding the approaches. It is essential that there is more cross-domain data provided, so that there is a wider statistical spectrum that contributes to sound statistical analysis of a given approach. The ability to draw correct conclusions, that takes into consideration the experimental limitations, such as time, resources and adequate developments for conducting trails, has therefore been demonstrated.

The *external validity*, which basically regards generalisation of findings, is only conditionally satisfied. The generalisability is limited by characteristics such as case studies originating from one company only, as in Event-B measurements case, or one application domain, like in case of the Simulink research. Moreover, small-sized developments may potentially endanger the external validity, as in the case of measurements for Progress Diagrams. It is premature to generalise the outcome of our research and assess artefacts presented in this thesis in a larger spectrum. It is due to the lack of multi-domain nature of the particular paths of the research and problems with replication of the investigations. However, the results of the research were described in our publications in such a way that they explicitly indicate to what degree the findings were to be generalised. Moreover, the representativeness factor of our work is noteworthy, since three out of five test cases were large-scale and complex rigorous software developments.

8.2 Directions for Future Work

The directions for future work are to some extent determined by the threats to the validity of the presented approach, which we described earlier. Moreover, we feel that solutions presented in this manuscript can be further developed and extended.

Firstly, there is a need to create a tool support for the Progress Diagrams and related measurements. The tool for drawing the diagrams, as well as automatically generating a new refinement would be a part of the Rodin platform toolset related to UML-B plug-in. The visual part of the Progress Diagram would be composed of a reduced UML-B State machine diagram, whereas the tabular part with a compact view of the refinement properties should be automatically generated from refinement properties of an UML-B model. The Progress Diagram plug-in could also support instantiation of generic refinement patterns and possibly help in further identification and differentiation of patterns in refinement steps. Moreover, a separate functionality for the Progress Diagram plug-in should be created for automatic data collection, computation of metrics and reporting. This would reduce the risk of human error in gathering the data and significantly decrease the threat to the construct validity of presented approaches.

The next step to be taken is to continue collecting the data from future developments, e.g. case studies, and, if possible, perform more rigorous investigations, i.e. using formal experiments. It would be beneficial and interesting to explore the applicability of presented metrics and measurement plans to systems originating from other domains, but retaining the development

setting and approach. It is possible that the metrics and measures presented will have to be fine-tuned or refined in order to be meaningful when employed in other application areas.

By investigating more large-scale, complex and rigorous developments, the representativeness of collected data and generalisability of results would be achieved. The assumptions about meaningfulness of measurement models would also then be confirmed. Since the limitations of the approach are already known, they open possibilities of further investigation not only to tackle the known issues, but also to progress the work presented in this thesis.

Finally, it would be beneficial to discuss the collected data as supplementary evidence of the impact of rigorous approaches on the quality of developments. This would open research discussion on the relationships between formal approaches and e.g. cost and effort related to the development. Future work could involve using collected data to create repository of facts and essentials, which are necessary to convince managers and developers about prerequisites and advantages that the application of formal methods entails.

In future research another possible path to take is to investigate how formal methods, such as Event-B, influence the quality of safety-critical products and how they can be utilised from the perspective of qualification and standardisation processes. Moreover, it would be interesting to see how professionals and organisations can benefit in this process. Although standardisation cannot substitute practical understanding, it gives essential knowledge about the evaluation strategy of the product within a domain, be it software, hardware or system. It also determines a certain manner of proceeding with the assessment and, through that, it might assist in cooperation when developing a product towards its certification.

There are numerous measures, metrics and measurement programs for certain application domains that are required by e.g. standards or stakeholders. The meaning of some of them overlaps, as they are used to asses the same feature of an artefact, even if at times from a different viewpoint. Our conjecture is that it would be beneficial to group them and categorise them according to their representation and possible relevance for specific domains. This suggests establishing an interactive framework for metrics and measurements, which includes their description and proposes a selection of application tactics for each of the metric. This initiative should recommend an off-the-shelf metrics for particular development settings, development types and phases. It should be based on the evaluation perspective, e.g. managerial or developers'. The metrics that would be contained in the framework should maintain SMARTER characteristics [112], i.e. they should be specific, measurable, actionable, relevant and timely, but also easy to manage and reusable.

8.3 Concluding Remarks

Our publications target research and industry domains. In particular, book chapters (Papers 1 and 2), and publications for conferences within the International Software Quality Institute (Papers 3 and 4) represent these two categories, respectively. They impact the contemporary knowledge and narrow the gap between research and industry-oriented environments. They propose the innovative metrics and measurements viewpoint on rigorous developments and shift the acquired knowledge beyond the university boundaries.

The research in Paper 5 not only continues the work initiated in Paper 4, but also focuses on higher-level aspects of measurements, i.e. the transferability of metrics between the settings and domains. The last paper included in this thesis (Paper 6) was acknowledged by software quality professionals at the World Congress for Software Quality. It confirms the perception from previous work that feasible metrics share certain characteristics, e.g. relevance, manageability or reusability.

We strongly believe that our contribution advances the state of research in the academic community and enables its transfer to non-academic organisations. Furthermore, fostering the mission of metrics as means to achieve valuable feedback on aspects of quality is essential in our work. It is our ambition for this thesis to promote measurements as an essential part of quality control and a strategy towards quality improvement.

Bibliography

[1] International Electrotechnical Commission (IEC), *IEC 61508 1-7 - Functional Safety of Electrical, Electronic, Programmable Electronic Safety-related Systems*. International Electrotechnical Commission, 2010.

[2] Jean-Raymond Abrial, "Extending B without Changing it (for Developing Distributed Systems)", in *Proceesings of 1st Conference on the B Method*, Nantes, 1996.

[3] Jean-Raymond Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge: Cambridge University Press, 2010.

[4] Jean-Raymond Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] Jean-Raymond Abrial, "User Manual of the RODIN Platform, Version 2.3", 2009.

[6] Nuno Amalio, Susan Stepney, and Fiona Polack, "Formal Proof from UML Models", in *Formal Methods and Software Engineering*, Heidenberg, 2004.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[8] Ralph-Johan Back, *On the Correctness of Refinement Steps in Program Development*. Åbo Akademi, Department of Computer Science, 1978.

[9] Ralph-Johan Back, *Refinement Calculus, Part II: Parallel and reactive programs. Stepwise Refinement of Distributed Systems*. Springer-Verlag, 1990.

[10] Ralph-Johan Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control", in *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1983, pp. 131-142.

[11] Ralph-Johan Back and Kaisa Sere, "From modular systems to action systems", in *Software - Concepts and Tools 17*, 1996, pp. 26-39.

[12] Ralph-Johan Back and Joakim von Wright, *Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science*. Heidelberg: Springer, 1998.

[13] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig, "Software complexity and maintenance costs", *Communications of the ACM*, vol. 36, pp. 81-94, 1993.

[14] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock, "Quality Attributes", Pittsburgh, Pennsylvania, CMU/SEI-95-TR-021, 1995.

[15] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach, "The Goal Question Metric Approach", in *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc, 1994, pp. 528-532.

[16] Robin Bloomfield, Dan Craigen, Frank Koob, Markus Ullmann, and Stefan Wittmann, "Formal Methods Diffusion: Past Lessons and Future Prospects", in *19th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, Rotterdam, 2000.

[17] Barry W. Boehm, J. R. Brown, and J. R. Kaspar, *Characteristics of Software Quality*. North Holland, 1978.

[18] Egon Boerger and Robert Staerk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[19] Pontus Boström, *Formal design and verification of systems using domain-specific languages. Ph D thesis*. Turku, Finland: Turku Centre for Computer Science, 2008, Ph. D thesis.

[20] Pontus Boström, Richard Grönblom, Tatu Huotari, and Jonatan Wiik, "An Approach to Contract-Based Verification of Simulink Models", Turku, TUCS Technical Report Number 985, 2010.

[21] Pontus Boström, Lionel Morel, and Marina Waldén, "Stepwise Development of Simulink Models Using the Refinement Calculus Framework", in *Theoretical Aspects of Computing (ICTAC2007)*, Macao, 2007.

[22] Jonathan P. Bowen and Michael G. Hinchey, "Ten Commandments of Formal Methods", *Computer*, vol. 28 (4), pp. 56-63, 1995.

[23] Jonathan P. Bowen and Michael G. Hinchey, "Ten Commandments of Formal Methods Ten Years Later", *IEEE Software*, vol. 39 (1), pp. 40-48, 2006.

[24] Jonathan Bowen and Victoria Stavridou, "The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective", in *Industrial-Strength Formal Methods*, Heidelberg, 1993.

[25] Lionel C. Briand and Sandro Morasca, "Software Measurement and Formal Methods: A Case Study Centered on TRIO+ Specifications", in *First International Conference on Formal Engineering Methods (ICFEM '97)*, Hiroshima, 1997.

[26] Frederick P. Brooks Jr., "No Silver Bullet - Essence and Accidents of Software Engineering", in *IFIP Tenth World Computing Conference*, Amsterdam, 1986.

[27] Manfred Bundschuh and Carol Dekkers, *The Measurement Compendium. Estimating and benchmarking Success with Functional Size Measurement.* Springer, 2008.

[28] Murray Cantor. (2001, JUly) Dr.Dobbs - The World of Software Development. [Online]. http://drdobbs.com/184415683

[29] David N. Card and Robert L. Glass, *Measuring Software Design Quality*. Prentice Hall, 1990.

[30] Ben Chelf and Andy Chou, “Controlling Software Complexity. The Business Case for Static Source Code Analysis”.

[31] Edmund C. Clarke and Jeanette M. Wing, “Formal Methods: State of the Art and Future Directions”, *ACM Computing Surveys* , vol. 28, pp. 626-643, 1996.

[32] Robert Cole, Sandeep Purao, Matti Rossi, and Maung K. Sein, “Being Proactive: Where Action Research meets Design Research”, in *Proceedings of International Conference on Information Systems ICIS 2005*, Las Vegas, 2005.

[33] James C. Connell, “Why Software Engineering Is Not B.S.”, *Dr. Dobb's Journal*, July 2001.

[34] Dan S.G. Craigen, Susan L. Gerhart, and Ted Ralston, “Formal methods technology: Impediments and innovation”, in *Applications of Formal Methods*. Prentice Hall, 1995, pp. 399-419.

[35] Dan S. G. Craigen and Ted Ralston, *An International Survey of Industrial Applications of Formal Methods. vol. 1, Purpose, Approach, Analysis and Conclusions.* U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, 1993.

[36] Marco D’Ambros, Alberto Bacchelli, and Michele Lanza, “On the Impact of Design Flaws on Software Defects”, in *10th International Conference on Quality Software*, Zhangjiajie, China, 2010.

[37] (2008, February) DEPLOY Project - Industrial deployment of system engineering methods providing high dependability and productivity. [Online]. http://www.deploy-project.eu/

[38] Edsger W. Dijkstra, “A Constructive Approach to the Problem of Program Correctness”, *BIT Nmerical Mathematics*, vol. 8(3), pp. 174-186, 1968.

[39] El Miloudi El Koursi and Gonzalo A. Mariano, “Assessment and certification of safety critical software”, in *Proceedings of the 5th Biannual World Automation Congress,* Orlando, 2002.

[40] (2008) Event-B.org. [Online]. http://www.event-b.org/index.html

[41] Tibor Farkas, Christian Hein, and Tom Ritter, “Automatic Evaluation of Modelling Rules and Design Guidelines”, in *From code centric to model centric software engineering: Practices, Implications and ROI.*, Enschede, 2009.

[42] Norman Fenton, “Software Measurement: A Necessary Scientific Basis”, *IEEE Transactions on Software Engineering*, vol. 20, pp. 199-206, 1994.

[43] Norman E. Fenton and Tracy Hall, “Implementing effective software metrics programs”, *Software IEEE*, pp. 55-65, 1997.

[44] Norman E. Fenton and A. A. Kaposi, “An engineering theory of structure and measurement”, in *Software Metrics. Measurement for Software Control and Assurance*. Elsevier, 1989, pp. 27-62.

[45] Norman E. Fenton and Martin Neil, “Software metrics: roadmap”, in *Conference on the Future of Software Engineering*, Limerick, Ireland, 2000.

[46] Norman E. Fenton and Shari Lawrence Pfleeger, *Software Metrics. A Rigorous & Practical Approach*. PWS Publishing Company, 1997.

[47] John Fitzgerald and Peter Gorm Larsen, “Balancing Insight and Effort: The Industrial Uptake of Formal Methods”, in *Formal Methods and Hybrid Real-Time Systems*, Heidelberg, 2007.

[48] Benoit Fraikin, Marc Frappier, and Regine Laleau, “State-based versus event-based specifications for information systems: a comparison of B and EB3”, *Software Systems Modelling*, vol. 4, pp. 236-257, 2005.

[49] Martin D Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi, “Strategies for incorporating formal specifications in software development”, *Communications of the ACM*, vol. Volume 37 (10), pp. 74-86, 1994.

[50] Martin D. Fraser and Vijay K. Vaishnavi, “A Formal Specifications Maturity Model”, *Communication of the ACM*, vol. 40 (12), pp. 95-103, 1997.

[51] Robert D. Galliers, Yasmin Merali, and Laura Spearing, “Manging Information Technology? How British Executives Perceive the Key Issues”, *Journal of Infomation Technology*, vol. 9, pp. 223-238, 1994.

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 416.

[53] Graphical Modeling Framework. [Online]. http://www.eclipse.org/modeling/gmp/

[54] Software Quality Metrics Methodology Working Group, *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Computer Society, 1998 / 2004.

[55] Yuri Gurevich, “A New Thesis”, *American Mathematical Society*, p. 317, 1985.

[56] Yuri Gurevich, “Reconsidering Turing’s Thesis: Toward More Realistic Semantics of Programs”, Michigan, CRL-TR-36-84, 1984.

[57] Anthony Hall, “Seven Myths of Formal Methods”, *IEEE Software*, pp. 11-19, 1990.

[58] Maurice Howard Halstead, *Elements of Software Science*. Elsevier North Holland, 1977.

[59] Peter G. Hamer and Gillian D. Frein, “M. H. Halstead's Software Science - A Critical Examination”, in *Proceedings, 6th International Conference on Software Engineering, ICSE*, Tokyo, 1982, pp. 184-199.

[60] Khairuddin Hashim and Elizabeth Key, "A Software Maintainability Attributes Model", *Malaysian Journal of Computer Science*, vol. 9 (2), pp. 92-97, 1996.

[61] Leo Hatvani, Anton Jansen, Christina Seceleanu, and Paul Pettersson, "An Integrated Tool for Trade-off Analysis of Quality-of-Service Attributes", in *Proceedings of The 2nd International Workshop on the Quality of Service-Oriented Software Systems*, Oslo, Norway, 2010.

[62] Duncan Haughey. Project Smart - Project Management Templates, Articles and Events. [Online]. http://www.projectsmart.co.uk/benefits-of-good-user-requirements.html

[63] Ian J. Hayes and Brendan P. Mahony, "Using Units of Measurement in Formal Specifications", *Formal Aspects of Computing*, vol. 7, pp. 329-347, 1995.

[64] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram, "Design science in information systems research", *MIS Quarterly*, vol. 28(1), pp. 75–105, 2004.

[65] Michael G. Hinchey, "Confessions of a formal methodist", in *SCS '02 Proceedings of the seventh Australian workshop conference on Safety critical systems and software*, Darlinghurst, 2003.

[66] John E. Hopcroft and Jeffrey Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[67] Arvind Hosagrahara. (2006, April) The Modeling Metric Tool. [Online]. http://www.mathworks.com/matlabcentral/fileexchange/5574

[68] Arvind Hosagrahara and P. Paul Smith, "Measuring Productivity and Quality in Model-Based Design", MATLAB Digest,Volume 14, Number 2 , March 2006. [Online]. http://www.mathworks.com/company/newsletters/digest/2006/mar/measuringprod.html

[69] Pei Hsia, "Learning to Put Lessons into Practice", *IEEE Software*, pp. 14-17, 1993.

[70] Mikko Huova et al., "Controller Design of Digital Hydraulic Flow Control Valve", in *The 11th Scandinavian International Conference on Fluid Power, SICFP'09*, Linköping, Sweden, 2009.

[71] Alexei Iliasov, "Model critic: detecting antipatterns in Event-B development", Turku, 5th December 2010.

[72] Alexei Iliasov and Alexander Romanovsky, "Refinement Patterns for Fault Tolerant Systems", in *Proceedings of Seventh European Dependable Computing Conference, EDCC-7,* Kaunas, 2008.

[73] Alexei Iliasov, Alexander Romanovsky, and Fernando Luis Dotti, "Structuring Specifications with Modes", in *Fourth Latin-American Symposium on Dependable Computing*, Joao Pessoa, 2009.

[74] Alexei Iliasov et al., “Supporting Reuse in Event B Development: Modularisation Approach”, in *Abstract State Machines, Alloy, B and Z. Second International Conference, ABZ 2010*, Orford, 2010.

[75] ISO/FDIS 26262 Road Vehicles - Functional Safety. [Online]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=43464

[76] Daniel Jackson, “A Direct Path to Dependable Software”, *Communications of the ACM*, pp. 78-88, 2009.

[77] Daniel Jackson, “Dependable Software by Design”, , May 2006.

[78] Daniel Jackson and Eunsuk Kang, “Separation of Concerns for Dependable Software Design”, in *Workshop on the Future of Software Engineering Research (FoSER)*, Santa Fe, 2010.

[79] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Peter Fennema, “Model Checking for Managers”, in *6th International SPIN Workshop on Practical Aspects of Model Checking*, Toulouse, 1999.

[80] Pertti Järvinen, “Action Research is Similar to Design Science”, *Quality & Quantity*, vol. 41, pp. 37-54, 2007.

[81] Stephen H. Kan, Victor R. Basili, and Larry N. Shapiro, “Software Quality: An Overview from the Perspective of Total Quality Management”, *IBM Systems Journal*, vol. 33, pp. 4-18, 1994.

[82] Barbara A. Kitchenham et al., “Preliminary Guidelines for Empirical Research in Software Engineering”, *IEEE Transactions on Software Engineering*, vol. 28, pp. 721-734, 2002.

[83] Barbara A. Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger, “Case Studies for Method and Tool Evaluation”, *IEEE Software*, vol. 12(4), pp. 52-62, 1995.

[84] John C. Knight, “Safety Critical Systems: Challenges and Directions”, in *24th International Conference on Software Engineering (ICSE 2002)*, Orlando, 2002.

[85] Peter B. Ladkin and Martyn Thomas. (2009, June) Formal Methods in Modern Critical-Software Development. [Online]. http://www.abnormaldistribution.org/2009/06/22/formal-methods-in-modern-critical-software-development/

[86] Jean C. Laprie, *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1991.

[87] Thierry Lecomte, “Applying a Formal Method in Industry: A 15-Year Trajectory”, in *Formal Methods for Industrial Critical Systems*, Eindhoven, 2009.

[88] Thierry Lecomte, Thierry Servat, and Guilhem Pouza, “Formal Methods in Safety-Critical Railway Systems”, in *10th Brasilian Symposium on Formal Methods*, Ouro Preto, 2007.

[89] B. Lees and David G. Jenkins, "Supporting Software Quality in an Integrated Safety-Critical Systems Development Environment", *Software Quality Journal*, vol. 5, pp. 117-125, 1996.

[90] Vladimir Mandic, L. Harjumaa, J. Markkula, and Markku Oivo, "Early Empirical Assessment of the Practical Value of GQM+Strategies", in *International Conference on Software Process, ICSP 2010*, Paderborn, 2010.

[91] Robert C. Martin, "OO Design Quality Metrics. An Analysis of Dependencies", 1994.

[92] (1994-2011) MathWorks - MATLAB and Simulink for Technical Computing. [Online]. http://www.mathworks.com/

[93] MAAB MathWorks Automotive Advisory Board, *Control Algorithm Modeling Guidelines Using Matlab®, Simulink®, and Stateflow®, Version 2.0.*, 2007.

[94] The MathWorks. Consulting the Model Advisor. [Online]. http://www.mathworks.com/help/toolbox/simulink/ug/f4-141979.html

[95] The MathWorks. Display diagnostic information about Simulink system. [Online]. http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/sldiagnostics.html

[96] MathWorks. Masked Subsystem Example, Simulink, R2010b Documentation. [Online]. http://www.mathworks.com/help/toolbox/simulink/ug/bsp2zlg-1.html

[97] The MathWorks. Simulink Design Verifier. [Online]. http://www.mathworks.com/products/sldesignverifier/

[98] The MathWorks. Simulink Verification and Validation Toolbox. [Online]. http://www.mathworks.com/products/simverification/

[99] Thomas McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, vol. Se-2, No. 4, pp. 308-320, 1976.

[100] Jim A. McCall, Paul K. Richards, and Gene F. Walters, "Factors in Software Quality", TR-77-369, Vols I, II, III, 1977.

[101] Steve McConnell, *Code Complete: A Practical Handbook of Software Construction, 2nd edition*. Microsoft Press, 2004.

[102] Mark S. Merkow and Lakshmikanth Raghavan, "Software security for developers", *Computerworld*, Sept. 2010.

[103] Christophe Metayer, Jean-Raymond Abrial, and Laurent Voisin, "Event-B Language, RODIN Deliverable 3.2 (D7)", 2005.

[104] Bertrand Meyer, "Dependable Software", in *Dependable Systems: Software, Computing, Networks*. Heidelberg: Springer-Verlag, 2006.

[105] Bertrand Meyer, “Design by Contract”, TR-EI-12/CO, 1986.

[106] Bertrand Meyer, “Design by Contract”, in *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1991, pp. 1-50.

[107] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[108] Steven P. Miller, “The Industrial Use of Formal Methods: Was Darwin Right?”, in *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, 1998.

[109] Carroll Morgan, *Programming from Specifications*. Prentice Hall, 1990.

[110] Gabriela Nicolescu and Pieter J. Mosterman, *Model-Based Design for Embedded Systems*. CRC Press, 2009.

[111] Fred Niederman, James C. Brancheau, and James C. Wetherbe, “Information Systems Management Issues for the 1990s”, *MIS Quarterly*, pp. 475-500, 1991.

[112] Marta Olszewska (Pląska), “SMARTER Metrics”, in *5th World Congress on Software Quality*, Shanghai, China, 2011.

[113] OMG, *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3*. OMG, 2010.

[114] Akira K. Onoma and Tsuneo Yamaura, “Practical Steps Toward Quality Development”, *IEEE Software*, pp. 68-76, 1995.

[115] David Lorge Parnas, “Really Rethinking 'Formal Methods'“, *IEEE Computer Society*, vol. 10, pp. 28-34, 2010.

[116] Mark C. Paulk, Bill Curtis, and Mary Beth Chrissis, “Capability Maturity Model, version 1.1”, *IEEE Software*, pp. 68-76, 1995.

[117] Marija Petkovic, Mark van den Brand, Alexander Serebrenik, and Elena Korshunova, “Computing System Metrics through Reverse Engineering”, in *Setting Quality Standards. Proceedings of the 11th International Conference of Quality Engineering in Software Technology (CONQUEST)*, Potsdam, Germany, 2008.

[118] Shari Lawrence Pfleeger and Les Hatton, “Investigating the Influence of Formal Methods”, *IEEE Computer*, vol. February 1997, pp. 33-43, 1997.

[119] Marta Pląska and Marina Waldén, “Quality Comparison and Evaluation of Digital Hydraulic Control Systems”, Turku, 2007.

[120] Rozilawati Razali, Colin F. Snook, and Michael R. Poppleton, “Comprehensibility of UML-based Formal Model – A Series of Controlled Experiments”, in *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, 2007.

[121] (2006) Rodin Platform. [Online]. http://www.event-b.org/platform.html or directly at http://sourceforge.net/projects/rodin-b-sharp/

[122] Peter H. Rossi, Mark W. Lipsey, and Howard E. Freeman, *Evaluation: A systematic approach (7th Ed.)*. Thousand Oaks, 2004.

[123] Research Triangle Park RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing", 2002.

[124] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 1999.

[125] John Rushby, "Formal Methods and the Certification of Critical Systems", Menlo Park, CA, 1993.

[126] Chris Sauer, D. Ross Jeffery, Lesley Land, and Philip Yetton, "The Effectiveness of Software Development Technical Reviews: A Behaviourally Motivated Program of Research", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 26, pp. 1-14, 2000.

[127] Bernhard Schaetz, Alexander Pretschner, Franz Huber, and Jan Philipps, "Model-Based Development of Embedded Systems", in *Advances in Object-Oriented Information Systems*, Heidenberg, 2002.

[128] (2000) Self Study Online Training. [Online]. http://www.exforsys.com/tutorials/uml/why-uml.html

[129] Bran Selic, "The Pragmatics of Model-Driven Development", *IEEE Software*, vol. 20(5), pp. 19-25, 2003.

[130] Richard Sharpe, "Formal Methods start to add up again", *Computing*, vol. 8th January 2004 issue, pp. 24-25, 2004.

[131] Renato Silva, Carine Pascal, T. S. Hoang, and Michael Butler, "Decomposition Tool for Event-B", in *Rodin User and Developer Workshop*, Düsseldorf, 2010.

[132] (2009) Simulink - Simulation and Model-Based Design. [Online]. http://www.mathworks.com/products/simulink/

[133] Colin Snook and Michael Butler, "UML-B and Event-B: an integration of languages and tools", in *International Conference on Software Engineering (SE2008)*, Innsbruck, 2008, pp. 336-341.

[134] Colin Snook and Michael Butler, "UML-B: Formal modelling and design aided by UML", *ACM Transactions on Software Engineering and Methodology*, vol. 15(1), pp. 92-122, 2006.

[135] Colin Snook and Marina Waldén, "Refinement of Statemachines using Event B semantics", in *Formal Specification and Development in B*, Besançon, 2007, pp. 171-185.

[136] “Software Errors Cost U.S. Economy $59.5 Billion Annually”, Gaithersburg, 2002.

[137] Bill St. Clair, “Growing Complexity Drives Need for Emerging DO-178C Standard”, *COTS Journal*, 2009.

[138] Iinternational Organization for Standardization, *ISO 9126-1 - Software engineering - Product quality — Part 1: Quality model*. ISO, 2001.

[139] International Organization for Standardization, *ISO/IEC 15504 (SPICE) Software Process Improvement and Capability Determination*.

[140] Dave Steinberg, Frank Budinsky, Marcelo Paternost, and Ed Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[141] Jagadish Suryadevara, Christina Seceleanu, and Paul Pettersson, “Pattern-driven Support for Designinig Component-based Architectural Models”, in *18th IEEE International Conference on Engineering of Computer-Based Systems (ECBS)*, Las Vegas, 2011.

[142] (2001) The 61508 Association. [Online]. http://www.61508.org/

[143] (2010, May) UML 2.3. [Online]. http://www.omg.org/spec/UML/

[144] UML-B plugin. [Online]. http://sourceforge.net/projects/rodin-b-sharp/files/Plugin_%20UML-B/

[145] Vijay K. Vaishnavi and Martin D. Fraser, “A Validation Framework for a Maturity Measurement Model for Safety-Critical Software Systems”, in *ACM-SE 36 Proceedings of the 36th annual Southeast regional conference*, New York, 1998, pp. 314-322.

[146] Marcel F. van Amstel and Mark G.J. Brand, “Model transformation analysis: staying ahead of the maintenance nightmare”, in *In Cabot, J. & Visser, E. (Eds.), Theory and Practice of Model Transformations, Proceedings of 4th International Conference on Model Transformations (ICMT 2011)*, Zurich, 2011.

[147] Marcel F. van Amstel and Mark G.J. Brand, “Using metrics for assessing the quality of ATL model transformations”, in *In M. Tisi, D. Wagelaar & I. Kurtev (Eds.), Proceedings of the 3rd International Workshop on MOdel Transformation with ATL (MtATL 2011) in conjunction with TOOLS 2011 Federated Conferences*, Zurich, 2011.

[148] Hans van Leunen. (2010, October) History Of A War Against Software Complexity. [Online]. http://www.science20.com/quaternion_waltz/history_war_against_software_complexity

[149] Rini Van Solingen and Egon Berghout, *The Goal/Question/Metric Method (GQM). A practical method for quality improvement of software development.* McGraw-Hill Inc., 1999.

[150] Rick Vinter, Martin Loomes, and Diana Kornbrot, "Applying Software Metrics to Formal Specifications: A Cognitive Approach", *IEEE International Symposium on Software Metrics*, pp. 216-223, 1998.

[151] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications , 2006.

[152] Marina Waldén and Kaisa Sere, "Reasoning about Action Systems using the B-Method", *Formal Methods in System Design*, vol. 13, pp. 5-35, 1998.

[153] Humphrey S. Watts, "The Software Quality Challenge", *The Journal of Defense Software Engineering*, pp. 4-9, 2008.

[154] Robin W. Whitty, "Research in Specification Metrics", *IEEE Colloquium on Software Metrics*, pp. 2/1 - 2/2, 2002.

[155] Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14(4), pp. 221-227, 1971.

[156] Claes Wohlin et al., *Experimentation in Software Engineering: An Introduction*. Heidelberg: Springer, 2000.

[157] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and Fitzgerald John, "Formal Methods: Practice and Experience", *ACM Computing Surveys*, vol. 41, pp. No. 4, Article 19, 2009.

[158] Fangjun Wu and Tong Yi, "Measuring Z Specifications", *ACM SIGSOFT Software Engineering Notes*, vol. 29 (5), pp. 1-5, 2004.

[159] Amid Yadav and Rasul A. Khan, "Measuring Design Complexity – An Inherited Method Perspective", *SIGSOFT Software Engineering Notes*, vol. 34, pp. 1-5, 2009.

[160] Robert K. Yin, *Case study research—Design and methods, 3rd ed.* Sage Publications, 1994.

[161] Edward Nash Yourdon, "Quality: What It Means and How to Achieve It", *Management Information Science*, pp. 43-47, 1993.

[162] Yujun Zheng, Kan Wang, and Jinyun Xue, "An extension of COCOMO II for the B-Method", in *Economics-Driven Software Engineering Research*, Shanghai, 2006.

[163] Horst Zuse, *Software Complexity: Measures and Methods*. Walter de Gruyter & Co, 1990.

Part II - Research Publications

Publication 1

Documenting the Progress of the System Development

Marta Pląska, Marina Waldén and Colin Snook

Originally published in:
The Book on Methods, Models and Tools for Fault Tolerance. A. Romanovsky, M. Butler, C. Jones, and E. Troubitsyna, (Eds), LNCS 5454, Springer-Verlag, Heidelberg, 2009.

Based on the publication:
Marta Pląska, Marina Waldén and Colin Snook, Documenting the Progress of the System Development, In Proceedings of Workshop on Methods, Models and Tools for Fault Tolerance, Oxford (United Kingdom), July 2007.

Extended abstract published in:
The Proceedings of Nineteenth Nordic Workshop on Programming Theory, Oslo (Norway), October 2007. (Marta Pląska, Marina Waldén and Colin Snook: "Visualising Program Transformations in a Stepwise Manner")

©2009 Springer Science + Business Media.
Reprinted with kind permission of Springer Science + Business Media.

Documenting the Progress of the System Development*

Marta Pląska[1], Marina Waldén[1] and Colin Snook[2]

[1] Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] University of Southampton, Southampton, SO17 1BJ, UK

Abstract. While UML gives an intuitive image of the system, formal methods provide the proof of its correctness. We can benefit from both aspects by combining UML and formal methods. Even for the combined method we need consistent and compact description of the changes made during the system development. In the development process certain design patterns can be applied. In this paper we introduce progress diagrams to document the design decisions and detailing of the system in successive refinement steps. A case study illustrates the use of the progress diagrams.

Keywords: Progress diagram, Statemachines, Stepwise development, Refinement, Refinement Patterns, UML, Event-B, Action Systems, Graphical representation.

1. Introduction

For complex systems the stepwise development approach of formal methods is beneficial, especially considering issues of ensuring the correctness of the system. However, formal methods are often difficult for industrial practitioners to use. Therefore, they need to be supported by a more approachable platform. The Unified Modelling Language (UML) is commonly used within the computer industry, but currently, mature formal proof tools are not available. Hence, we use formal methods in combination with the semi-formal UML.

For a formal top-down approach we use the Event B formalism [11] and associated proof tool to develop the system and prove its correctness. Event-B is based on Action Systems [4] as well as the B Method [1], and is related to B Action Systems [22]. With the Event-B formalism we have tool support for proving the correctness of the development. In order to translate UML models into Event B, the UML-B tool [18, 19] is used. UML-B is a specialisation of UML that defines a formal modelling notation combining UML and B.

The first phase of the design approach is to state the functional requirements of the system using natural language illustrated by various UML diagrams, such as statechart diagrams and sequence diagrams that depict the behaviour of the system. The system is built up gradually in small steps using superposition refinement [3, 10]. We rely on patterns in the refinement process, since these are the cornerstones for creating *reusable* and *robust* software [2, 7]. UML diagrams and corresponding Event

* Work done within the RODIN-project, IST-511599

B code are developed for each step simultaneously. To get a better overview of the design process, we introduce the *progress diagram*, which illustrates only the refinement-affected parts of the system and is based on statechart diagrams. Progress diagrams support the construction of large software systems in an incremental and layered fashion. Moreover, they help to master the complexity of the project and to reason about the properties of the system. We illustrate the use of the diagrams with a case study.

The rest of the paper is organised as follows. In Section 2 we give an overview of our case study, Memento, from a general and functional perspective. An abstract specification is presented as a graphical, as well as a formal representation in Section 3. Section 4 describes stepwise refinement of systems and gives refinement patterns and Section 5 introduces the idea of progress diagrams. The system development is analysed and illustrated with the progress diagrams relying on the case study in Section 6. The related work is presented in Section 7. We conclude with some general remarks and our future work in Section 8.

2. Case Study – Memento Application

The *Memento* application [14] that is used as a case study in this paper is a commercial application developed by *Unforgiven.pl*. It is an organiser and reminder system that has evolved into an internet-based application. Memento is designed to be a framework for running different modules that interact with each other.

In the distributed version of Memento every user of the application must have its own, unique identifier, and all communication is done via a central application server. In addition to its basic reminder and address book functions, Memento can be configured with other function modules, such as a simple chat module. Centralisation via the use of a server allows the application to store its data independently of the physical user location, which means that the user is able to use his own Memento data on any computer that has access to the network.

The design combines the web-based approach of internet communicators and an open architecture without the need for installation at client machines. During its start-up the client application attempts to *connect to a central server*. When the connection is established, the *preparation phase* begins. In this phase the user provides his/her unique identifier and password for authorisation. On successful login the server responds by sending the data for the account including a list of contacts, news, personal files etc. Subsequently the *application searches for modules* in a working folder and attempts to initialise them, so that the user is free to run any of them at any time. During execution of the application, *commands from the server and the user are processed* at once. Memento translates the requested actions of the user to internal commands and then handles them either locally or via the server. Upon a termination command Memento *finalises all the modules*, saves the needed data on the server, logs out the user and *closes the connection*. To minimise the risk of data loss, in case of fatal error, this termination procedure is also part of the fatal exception handling routine.

3. Abstract Specification

3.1. UML-models

We use the Unified Modelling Language™ (UML) [5], as a way of modelling not only the application structure, behaviour, and architecture of a system, but also its data structure. UML can be used to overcome the barrier between the informal industry world and the formal one of the researchers. It provides a graphical interface and documentation for every stage of the (formal) development process. Although UML offers miscellaneous diagrams for different purposes, we focus on two types of these in our paper: sequence diagrams and statechart diagrams.

The sequence diagram is used within the development of the system to show the interactions between objects and the order in which these interactions occur. The diagram can be derived directly from the requirements. Furthermore, it may give information on the transitions of the statemachines. The interaction between entities in the sequence diagram can be mapped to self-transitions on the statechart diagram to model communication between the modelled entity and its external entities.

In our case study the external entities are the server and the users interacting with the modelled entity Memento. An example of a sequence diagram for the application is given in Fig. 1, where part of the requirements (the emphasized text in Section 2) concerning the server connection and the program preparation phase is shown. In the diagram we describe the initialisation phase of the system, which consists of establishing a connection (in the connection phase) and then preparing the program (in the preparation phase). The first of these actions requires the interaction with the server through an internet connection. The second action requires communication with user as well. The described interaction (in Fig. 1) is transferred to a statechart diagram as transition *tryInit* (to later be refined to the transitions *tryConn* and *tryPrep* as explained in Section 6).

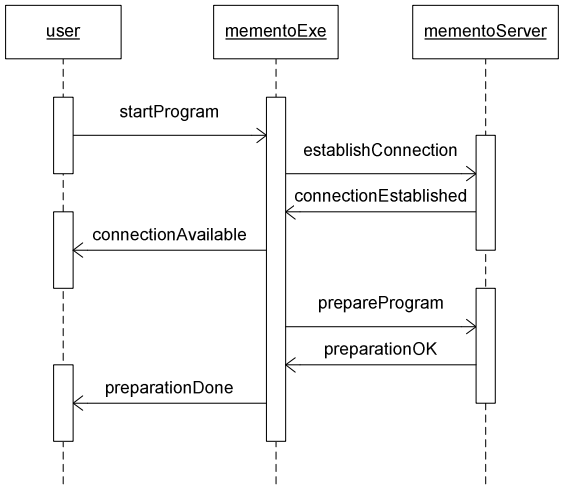


Fig. 1. Sequence diagram presenting the object interaction in the *initialisation phase*

In statechart diagrams objects consist of states and behaviours (transitions). The state of an object depends on the previous transition and its condition (guard). A statechart diagram provides a graphical description of the transitions of an object from one state to another in response to events [12, 13]. The diagram can be used to illustrate the behaviour of instances of a model element. In other words, a statechart diagram shows the possible states of the object and the transitions between them.

The statemachine depicting the abstract behaviour of Memento is shown in Fig. 2. The first phase is to initialise the system by communicating with the server. It is modelled with the event *tryInit*. When initialisation has been successfully completed, the transition *goReady* brings the system to the state ready, where it awaits and processes the user and server commands. Upon the command *close*, the system enters the finalisation phase, which leads to the system cleanup and proper termination.

The detection of errors in each phase is taken into consideration. In the model, the errors are captured by transitions targeting the suspended state (*susp*), where error handling (rollback) takes place. The system may return to the state where the error was detected, if the error happens to be recoverable. If the error is non-recoverable, the fatal termination action is taken and the system operation finishes. Any error detected during or after finalisation phase is always non-recoverable.

We use the following notation for the transitions in statechart diagrams and in the Event-B code in the rest of the paper. The symbols '::' and ':∈' stand for non-deterministic assignment and are applied interchangeably, in the diagrams and the code, respectively. The symbol ':=' is used in assignments, whereas '||' symbol denotes that the operands are executed concurrently. All of the mentioned symbols are placed in the statement parts. By the use of a junction pseudo state [20] (marked with angled brackets '<>') that denotes the old, refined transition, we indicate the refinement relation between new transitions and previous abstract ones.

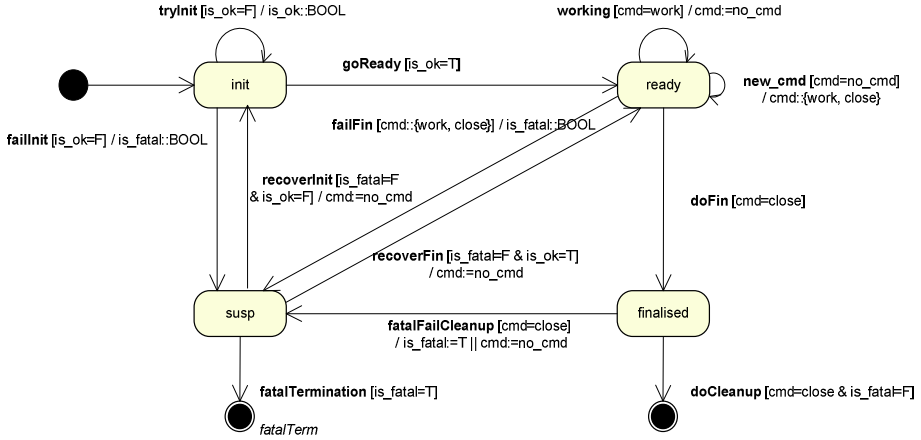


Fig. 2. The abstract statemachine of Memento

3.2. Formal Specification

In order to be able to reason formally about the abstract specification, we translate it to the formal language Event B [11]. An Event-B specification consists of a model and its context that depict the dynamic and the static part of the specification, respectively. They are both identified by unique names. The context contains the sets and constants of the model with their properties and is accessed by the model through the SEES relationship [1]. The dynamic model, on the other hand, defines the state variables, as well as the operations on these. Types and properties of the variables are given in the invariant. All the variables are assigned an initial value according to the invariant. The operations on the variables are given as events of the form **WHEN** guard **THEN** substitution **END** in the Event-B specification. When the guard evaluates to true the event is said to be enabled. If several events are enabled simultaneously any one of them may be chosen non-deterministically for execution. The events are considered to be atomic, and hence, only their pre and post states are of interest. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible [11].

Each transition of a statechart diagram is translated to an event in Event-B. Below we show the Event B-translation of the statemachine concerning the initialisation (state *init*) of the cooperation with the server in Fig. 2:

MACHINE Memento
SEES Data
VARIABLES is_fatal, is_ok, cmd, state
INVARIANT is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
(state=init ⇒ cmd=no_cmd) ∧ ...
INITIALISATION is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init
EVENTS
tryInit = **WHEN** state=init ∧ is_ok=FALSE **THEN** is_ok :∈ BOOL **END**;
failInit = **WHEN** state=init ∧ is_ok=FALSE **THEN** state:=susp || is_fatal :∈ BOOL **END**;
recoverInit= **WHEN** state=susp ∧ is_ok=FALSE ∧ is_fatal=FALSE **THEN** state:=init || cmd:=no_cmd **END**;
goReady = **WHEN** state=init ∧ is_ok=TRUE **THEN** state:=ready **END**;
...
END

The variables model a proper initialisation (*is_ok*), occurrence of a fatal error (*is_fatal*), as well as the command (*cmd*) and the state of the system (*state*). Initially no command is given and the initialisation phase is marked as not completed (*is_ok := FALSE*). The guards of the transitions in the statechart diagram in Fig. 2 are transformed to the guards of the events in the Event B model above, whereas the substitutions in the transitions are given as the substitutions of the events. The feasibility and the consistency of the specification are then proved using the Event-B prover tool.

4. System Refinement and Refinement Patterns

It is convenient not to handle all the implementation issues at the same time, but to introduce details of the system to the specification in a stepwise manner. Stepwise refinement of a specification is supported by the Event-B formalism. In the

refinement process an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. We use the superposition refinement technique [3, 11, 22], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. The variables are added gradually to the specification with their conditions and properties. The computation concerning the new variables is introduced in the existing events by strengthening their guards and adding new substitutions on these variables. New events, assigning the new variables, may also be introduced.

4.1. Refinement of the System

System *C* is said to be a correct refinement of *A* if the following proof obligations are satisfied [11, 20, 22]:

1. The initialisation in *C* should be a refinement of the initialisation in *A*, and it should establish the invariant in *C*.
2. Each old event in *C* should refine an event in *A*, and preserve the invariant of *C*.
3. Each new event in *C* (that does not refine an event in *A*) should only concern the new variables, and preserve the invariant.
4. The new events in *C* should eventually all be disabled, if they are executed in isolation, so that one of the old events is executed (non-divergence).
5. Whenever an event in *A* is enabled, either the corresponding event in *C* or one of the new events in *C* should be enabled (strong relative deadlock freeness).
6. Whenever an error detection event (event leading to the state *susp)* in *A* is enabled, an error detection event in *C* should be enabled (partitioning an abstract representation of an error type into distinct concrete errors during the refinement process [21]).

The tool support provided by Event-B allows us to prove that the concrete specification *C* is a refinement of the abstract specification *A* according to the Proof Obligations (1) - (6) given above.

4.2. Modelling Refinement Patterns

In order to guide the refinement process and make it more controllable, refinement patterns [12] can be applied. We are using the following notation for the patterns in the rest of the paper. A typical event consists of a guard *G(V)* and an action *S(V)*, where *G(V)* is some supplementary predicate on the variables *V,* often represented as a conjunction of several individual guards, and *S(V)* is some supplementary assignment of the variables *V*. The variables *V* are of a general type (TYPE). For instance, in the Event-B code for the refinement EX3c $G_i(y)$ denotes a guard on variable *y* of the general type TYPE, while $S_i(y)$ denotes some assignment of variable *y*.

In all the pattern diagrams (except in the choice paths in Fig. 5) we omit the guards on the transitions for better readability of the diagrams. The code added in the current refinement step is indicated by a darker background.

4.2.1 Refining the States

Let us first concentrate on the abstract specification given in Fig. 3a. It is pictured by a statechart diagram consisting of two states (*st1* and *st2*), a self transition *tr1* for the state *st1* and a transition *tr2* from state *st1* to the state *st2*. We are focusing on data refinement and event refinement patterns enabled by the data refinement. The former is shown in the statechart diagram in Fig. 3b (splitting states into substates and adding transitions between them), while an example of the latter is given in Fig. 3c (splitting existing transitions).

Splitting the states and adding new transitions are commonly performed in one refinement step. The two steps shown in Figures 3b and 3c are shown separately here only to depict the details of the complete data and event refinement. Generally, when refining the states, we want to add some new features/variables at the same time as we split the transitions.

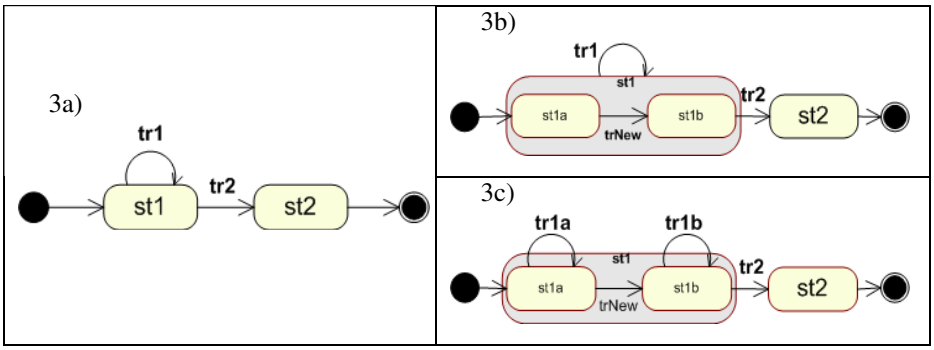


Fig. 3. Refinement patterns – basic data and event refinements

For the abstract specification depicted in Fig. 3a we have the following Event-B specification code:

MACHINE EX3a
VARIABLES state
INVARIANT state ∈ {st1, st2}
INITIALISATION state:=st1
EVENTS
tr1 = **WHEN** state=st1 **THEN** state:=st1 **END**;
tr2 = **WHEN** state=st1 **THEN** state:=st2 **END**
END

The Event-B code for the pattern concerning the data refinement, i.e. splitting the states, is illustrated in the Fig. 3b as follows:

REFINEMENT EX3b
REFINES EX3a
VARIABLES state, state1
INVARIANT state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b}
INITIALISATION state:=st1 || state1:=st1a
EVENTS
tr1 = **WHEN** state=st1 **THEN** state:= st1 **END**;
trNew = **WHEN** state=st1 ∧ state1=st1a **THEN** state1:=st1b **END**;
tr2 = **WHEN** state=st1 ∧ state1=st1b **THEN** state:=st2 **END**
END

The pattern for separating an existing transition (event refinement) corresponding to the diagram in Fig. 3c is as follows:

REFINEMENT EX3c
REFINES EX3a
VARIABLES state, state1, y, z
INVARIANT state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b} ∧ y∈TYPE ∧ z∈TYPE ∧ I(y,z)
INITIALISATION state:=st1 || state1:=st1a || y:∈TYPE || z:∈TYPE
EVENTS

tr1a (refines tr1) =	**WHEN** state=st1 ∧ state1=st1a ∧ $G_{1a}(y)$ **THEN** state:=st1 || state1:=st1a || $S_{1a}(y)$ **END;**
tr1b (refines tr1) =	**WHEN** state=st1 ∧ state1=st1b ∧ $G_{1b}(z)$ **THEN** state:=st1 || state1:=st1b || $S_{1b}(z)$ **END;**
trNew =	**WHEN** state=st1 ∧ state1=st1a ∧ $G_N(y)$ **THEN** state1:=st1b **END;**
tr2 =	**WHEN** state=st1 ∧ state1=st1b ∧ $G_2(z)$ **THEN** state:=st2 **END**

END

Each of the refined event uses some of the new variables (y and z) in its guards ($G(y)$ and $G(z)$) and actions ($S(y)$ and $S(z)$) to reduce non-determinism. The guards $G_{1a}(y)$ and $G_n(y)$ are created in such a way that they guarantee progress. In the same manner $G_n(y)$ should imply $G_2(z)$ or $G_{1b}(z)$, moreover guards $G_2(z)$ and $G_{1b}(z)$ should also be formed to guarantee the progress of the system. When inserting conditions on new properties to the guards, the failure management is in general also refined in a corresponding manner in order to design a fault tolerant behaviour. Nevertheless, here we concentrate our patterns on the proper and desirable behaviour of the system. An example of another pattern of the basic data and event refinement including failure management is given by Snook and Waldén [20]. In that pattern a loop is created in the superstate.

In order to give an intuition of the correctness of the patterns, we state how the Proof Obligation Rules given in Section 4.1 are satisfied by the patterns. Moreover, the Proof Obligations hint at how problems in the program design can be detected more easily.

According to the Proof Obligations (1) and (2) in Section 4.1 the initialisation and the events are refined to take the new variables into consideration. The guards of the old events may be strengthened and assignments concerning the new variables added. During the system development we may also want to refine an existing event by splitting it into several separate events. As acknowledged in Proof Obligation (3), the new events are only permitted to assign the new variables, but may, however, refer to the old variables. The guard of the new event should be composed in such a way that the new event, together with the refined events, ensures progress of the system (Proof Obligation (5)).

The new events should not take over the execution (Proof Obligation (4)), which can be assured by disallowing the new transitions in the statemachine diagram (corresponding to new events in the Event-B model) from forming a loop. The Event-B prover requires an expression, called a 'variant', that gives a Natural number that is decreased by all of the new transitions between the substates. To deduce a suitable variant, graph theory is applied. The states of the statemachine representing the system are numbered according to the minimum path length to a refining transition. Hence, if the new transitions form a sequence that progresses towards a refining transition, the function that defines this numbering for each state will be strictly

decreased as the state changes. If loops are unavoidable in the new events, the auxiliary variables must be used in the variant in order to provide a suitable variant that decreases throughout the loop. Each new transition has to lead to a new state with a lower designated number or (in case of loops) alter the auxiliary state variables, thereby decreasing the variant. To avoid deadlock, a route from every new transition to one that refines an old transition should exist. This is a necessary, but not sufficient, condition for relative deadlock freeness. If there exists no sequence of new transitions which can reach one that refines an old transition, meaning there is no route, then new events terminate (without enabling an old event) and a new deadlock is introduced [20].

As properties are added to the system, the potential failure management should also be refined. If a fault appears at a substate it should be viable to return to that substate after recovery. This can be achieved by dividing an abstract failure into more specific failures on the new features in conformance with Proof Obligation (6). Note that new failure situations are not introduced in our case, as it is a general pattern that can be adjusted to the specific needs.

4.2.2 Flattening States

When refining the system by superposition and at the same time splitting the states in a hierarchical manner, we have to deal with the states that are nested in the superstate due to the consequent refinement steps. This development, although performed in a stepwise manner, at some point makes the system model unreadable. Therefore we apply the flattening pattern, which removes the most external superstate, leaving the substates intact.

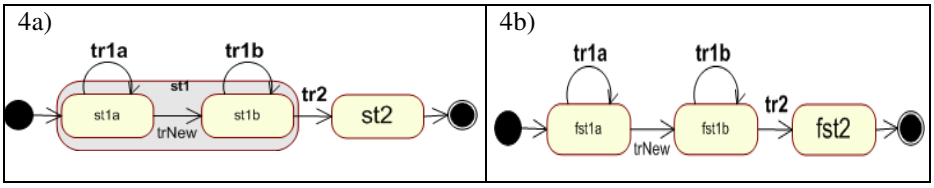


Fig. 4. Refinement pattern – flattening of the hierarchical states

In Fig. 4 we present the flattening pattern applied to the model from Fig. 3c. In Fig. 4a we model the hierarchical structure of states, i.e. state *st1* is a superstate for the states *st1a* and *st1b*. By applying the flattening pattern, we remove the superstate *st1*. This is possible, when giving an appropriate invariant preserving the relation between the states in the model, namely relating the states from the old model to the states in the new model. This is correlated with the change of the naming of the variables in order to preserve the invariant.

Note that flattening can only be performed once the parent state is neither the source nor target of any transitions. That is, other patterns should first be applied to move all the parents' transitions to its substates so that the parent state is completely redundant.

Here we show the Event-B code for the refined model:

REFINEMENT EX4b
REFINES EX4a
VARIABLES newState, y, z, v
INVARIANT newState ∈ {fst1a, fst1b, fst2} ∧ newState ∈ NEWSTATE
∧ y∈TYPE ∧ z∈TYPE ∧ v∈TYPE ∧ R_4(y,z,v) ∧
newState=fst1a ⇔ (state=st1 ∧ state1=st1a) ∧
newState=st1b ⇔ (state=st1 ∧ state1=st1b) ∧ state=st2 ⇒ newState=fst2 ∧
newState=fst2 ⇒ state=st2 ∧ state=st1 ⇒ (newState=fst1a ∨ fst1b)
INITIALISATION newState:=fst1a || v:∈TYPE || y:∈TYPE || z:∈TYPE
EVENTS
tr1a (refines tr1) = **WHEN** newState=fst1a ∧ G_{1a}(y) ∧ G_{1a}(z) ∧ G_{1a}(v)
THEN newState:=fst1a || S_{1a}(y) || S_{1a}(z) || S_{1a}(v) **END**;
tr1b (refines tr1) = **WHEN** newState=fst1b ∧ G_{1b}(y) ∧ G_{1b}(z) ∧ G_{1b}(v)
THEN newState:=fst1b || S_{1b}(y) || S_{1b}(z) || S_{1b}(v) **END**;
trNew = **WHEN** newState=fst1a ∧ G_{nr}(y) ∧ G_{nr}(z) ∧ G_{nr}(v)
THEN newState:=fst1b || S_{nr}(y) || S_{nr}(z) || S_{nr}(v) **END**;
tr2 = **WHEN** newState=fst1b ∧ G_{2r}(y) ∧ G_{2r}(z) ∧ G_{2r}(v)
THEN newState:=fst2 || S_{2r}(y) || S_{2r}(z) || S_{2r}(v) **END**
END

Since flattening the state hierarchy is rather a rewriting step than a refinement step, the proof obligations in Section 4.1 trivially hold. We rely on the invariant giving the relation between the flattened state and the hierarchical states.

4.2.3 Separating Existing Transitions – Choice Paths

In order to perform event refinement, particularly to separate existing events, we can split the transition into alternative paths using the black diamond-shaped choice symbol (salmiakki) [20], where each choice is responsible for a separate event. The guards on the events are strengthened by the choice points. Each choice represents a separate event whose guard includes the conjunction of all the segments leading up to that path. Thus, the guard enabling a given event is the conjunction of all the conditions of choice paths leading up to the choice point.

Fig. 5 illustrates a simple pattern for adding features to the specification and expanding its functionality. More specifically, the transition *tr1* is refined by two branches - transitions *tr1a* and *tr1b*. This could, for example, model the refinement of a non-deterministic event '*move*' to the more specific events '*move_forward*' and '*move_backward*'.

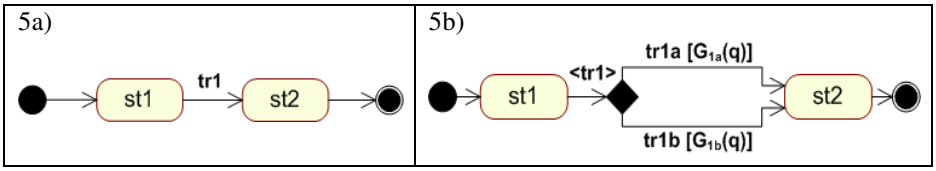


Fig. 5. Refinement pattern – event refinement: simple choice paths

The Event-B code corresponding to this pattern for the abstract machine is as follows:

MACHINE EX5a
VARIABLES state
INVARIANT state ∈ {st1, st2 }
INITIALISATION state:=st1
EVENTS
tr1 = **WHEN** state=st1 **THEN** state:=st2 **END**
END

The refinement can be expressed as the following Event-B machine:

REFINEMENT EX5b
REFINES EX5a
VARIABLES state, q
INVARIANT state ∈ {st1, st2 } ∧ q ∈ TYPE ∧ I(q)
INITIALISATION state:=st1 || q :∈ TYPE
EVENTS
tr1a (refines tr1) = **WHEN** state=st1 ∧ G_{1a}(q) **THEN** state:=st2 || S_{1a}(q)**END**;
tr1b (refines tr1) = **WHEN** state=st1 ∧ G_{1b}(q) **THEN** state:=st2 || S_{1b}(q) **END**
END

Guards of the transition *tr1* are strengthened via choice point, according to Proof Obligation (2). When splitting the transition *tr1* into two more specific transitions *tr1a* and *tr1b* we should ensure the progress of the system, fulfilling Proof Obligation (5).

With the choice paths pattern we can also show more detailed failure management. Fig. 6 depicts the creation of the choice path on the transition *tr1*, thus detailing the error detection. The transition *tr1* is refined by strengthening its guards concerning the new variable *q* ($G_1(q)$). The transition *tr2undef* is refined into transition *tr2undef2*, which stands for the detection of undetermined errors and *tr2a* which is modelling a particular type of failures ($\neg G_1(q)$).

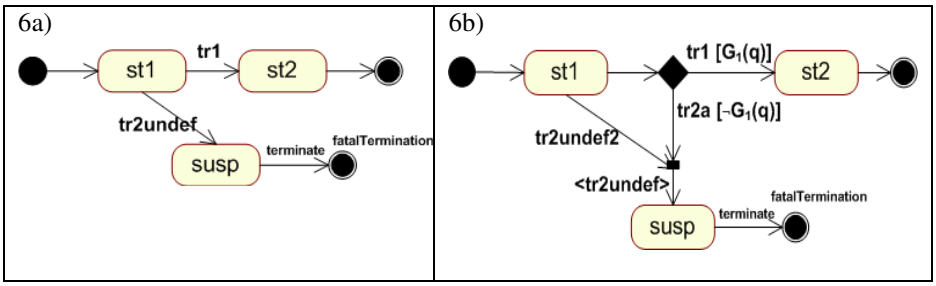


Fig. 6. Refinement pattern – event refinement: choice paths

The Event-B code for the abstract machine that we use as an example for choice paths pattern (event refinement) is as follows:

MACHINE EX6a
VARIABLES state
INVARIANT state ∈ {st1, st2, susp}
INITIALISATION state:=st1
EVENTS
tr1 = **WHEN** state=st1 **THEN** state:=st2 **END**;
tr2undef = **WHEN** state=st1 **THEN** state:=susp **END**
END

The refined model is expressed as an Event-B model given below:

REFINEMENT EX6b
REFINES EX6a
VARIABLES state, q
INVARIANT state ∈ {st1, st2, susp} ∧ q ∈ TYPE ∧ J(q)
INITIALISATION state:=st1 || state1:=st1a || q :∈ TYPE
EVENTS
tr1 = **WHEN** state=st1 ∧ G_1(q) **THEN** state:=st2 || S_1(q) **END**;
tr2undef2 (refines tr2undef) = **WHEN** state=st1 **THEN** state:=susp **END**;
tr2a (refines tr2undef) = **WHEN** state=st1 ∧ ¬G_1(q) **THEN** state:=susp || S_{2a}(q) **END**
END

We use the join (black bar symbol) to illustrate refinement of the failure transition *tr2undef*, which is split into two different failures, *tr2undef2* and *tr2a,* in accordance with Proof Obligation (6). The guard for transition *tr1* is strengthened (Proof Obligation (2)) by the conjunction of the negation of all the particular failures. In this way we can ensure that there will be an enabled event also in the refined model (Proof Obligation (5)).

4.2.4 Orthogonal Regions Pattern

Furthermore, we can also consider a pattern for adding the same behaviour to several states (orthogonal regions [20]) as a type of data and event refinement. This pattern can be used in case of architectural redundancy, i.e. when several states have incoming (entry) and outgoing (exit) transitions of similar functionality. Fig. 7 illustrates adding an orthogonal region (the lower region) to the superstate *susp*, which has new behaviour common to all the previous states (given in the higher region), applicable to all three kinds of failure. In order for the pattern to be correct, several conditions have to be fulfilled. The orthogonal region should not affect the mechanism of error detection. In Fig. 7 we show that the unnamed entry and exit transitions of the lower region connect to the named events of the upper region. It must be ensured that, when the new region is entered, at least one of the new transitions must be enabled (synchronisation condition). Moreover the exit transitions from the upper region are synchronised with equivalent transitions of the lower region, i.e., they are guarded by the lower region reaching a state that has an exit transition with which it can merge.

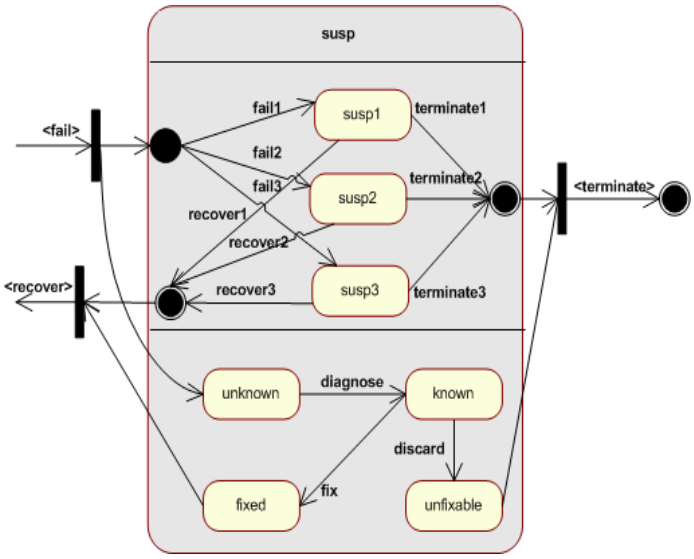


Fig. 7. Refinement pattern - superposition of an orthogonal region

Subsequently we give a machine and its refinement using the orthogonal states. The upper region of the state *susp* forms the abstract machine.

MACHINE EX7
VARIABLES state, suspState
INVARIANT state ∈ {susp, state_ok} ∧ suspState ∈ {susp1, susp2, susp3}
INITIALISATION state:=susp || suspState:={susp1, susp2, susp3}
EVENTS
fail1 = **WHEN** state=state_ok **THEN** state:=susp || stateSusp:=susp1 **END**;
recover1 = **WHEN** state=susp ∧ suspState=susp1 **THEN** state:=state_ok **END**;
terminate1 = **WHEN** state=susp ∧ suspState=susp1 **THEN** state:=fatalTermination **END**;
…
END

In the refinement the old state *susp* is composed with the orthogonal region.

REFINEMENT EX7a
REFINES EX7
VARIABLES state, suspState, ortState, s
INVARIANT state ∈ {susp, state_ok, fatalTermination} ∧ suspState ∈ {susp1, susp2, susp3} ∧ ortState ∈ {unknown, known, fixed, unfixable} ∧ s∈TYPE ∧ I(s)
INITIALISATION state:=susp || suspState:={susp1} || ortState:=unknown || s:∈ TYPE
EVENTS
fail1 = **WHEN** state=state_ok ∧ G_f(s) **THEN** state:=susp || stateSusp:=susp1 || ortState:=unknown **END**;
recover1 = **WHEN** state=susp ∧ suspState=susp1 ∧ G_r(s) ∧ ortState=fixed **THEN** state:=state_ok || S_r(s) **END**;
terminate1 = **WHEN** state=susp ∧ suspState=susp1 ∧ ¬G_f(s) ∧ ortState=unfixable **THEN** state:=fatalTermination || S_f(s) **END**;
…
diagnose = **WHEN** state=susp ∧ ortState=unknown **THEN** ortState:=known **END**;
discard = **WHEN** state=susp ∧ ortState=known **THEN** orState:=unfixable **END**;
fix = **WHEN** state=susp ∧ ortState=known **THEN** ortState:=fixed || S_f(s)**END**
END

As the events/transitions introduced in the orthogonal region are new events in the refinement, they should only concern new features to satisfy Proof Obligation (3). These new transitions should eventually hand over control to the old transitions, which is guaranteed by Proof Obligation (4). Hence, we need to generate a variant on the distance to an *exit* transition for these new transitions. In order to fulfil Proof Obligation (5) at least one of the new transitions (*diagnose* followed by *discard* or *fix*) must be enabled after entering the orthogonal region. This is caused by the fact that the recovering transitions and the terminating transitions wait to be enabled until the orthogonal region is prepared to synchronise with them. The composed events are then refinements of the old events, like for example *fail1*, *fail2*, *fail3,* in conformance with Proof Obligation (2). The orthogonal region strengthens the guards of the termination and recovery events, but at the same time it guarantees that an exit state of the orthogonal region will be reached (Proof Obligation (6)).

As the size of the system grows during the development, it is difficult to get a clear overview of the refinement process. In this paper we benefit from *progress diagrams* [16] to give an abstraction and graphical-descriptive view documenting the applied patterns in each step. The pattern types are illustrated in more detail with the progress diagrams to show the relevant development changes in a legible manner. This is of high importance especially when the system evolves into a significantly sized one.

5. Progress Diagrams

We exploit the progress diagram [16], which is in the form of a table divided into a description part and a diagram part. With this type of table we can point out the design patterns derived from the most important features and changes done in the refinement step. It provides compact information about each refinement step, thereby indicating and documenting the progress of the development. The tabular part briefly describes the relevant features or design patterns of the system in the development step. Moreover, it depicts how states and transitions are refined, as well as new variables that are added with respect to these features. Progress diagrams do not involve any mathematical notation and are, therefore, useful for communicating the development steps to non-formal methods colleagues.

Event-B classifies events as 'convergent' if they are new events that are expected to eventually relinquish control to an old (refined) event (i.e. it must decrease the variant). Events that are not convergent are classified as ordinary. A third classification, 'anticipating', refers to events that will be shown to be convergent in a future refinement, but we do not use such events in the examples of the patterns.

We call the transition that starts a sequence of convergent transitions an 'initiator'. To ensure feasibility of the sequence of transitions, the guard of an initiator must imply the guard of at least one of the old transitions that it could lead to. We call the transition 'refined' if it refines an existing transition according to the refinement rules (given in Section 4.1 of the paper), leaving the system in an equivalent state to the post-state of the transition being refined.

We envisage a tool which will automatically create a new refinement from the progress diagram. In order to be able to design and implement such a tool, we extend the tabular part of the progress diagram to provide the required information. As a result, we add new features in the Refined Transitions part indicating the source and target state of the refined transition, as well as the initialisation of the variables added in the current refinement step. The diagram part gives a supplementary view of the current refinement step and is, in fact, a fragment of the statechart diagram. It can be used as assistance for the developer and to support the documentation.

During the development we profit from the progress diagram, as we concentrate only on the refined part of the system. The combination of descriptive and visual approaches to show the development of the system gives a compact overview of the part that is the current scope of development. This enables us to focus on the details that we are most interested in, and provides a legible picture of the (possibly complex) system development. The visualisation helps us to better understand the refinement steps and proofs that need to be performed.

When proving the refined system, the progress diagram indicates the needed proof obligations. If new states (column "Ref. States") and variables (column "New Var.") are added, they should be initialised according to the invariant (Proof Obligation (1)). In the progress diagram the refined events are given in the column "Refined Transitions" and have a corresponding event in the column "Transitions" (Proof Obligation (2)). Also the convergent events are given in the column "Refined Transitions". However, they do not have a corresponding event in the column "Transitions". They may only assign the variables in column "New Variables" according to the invariant (Proof Obligation (3)). Furthermore, the non-divergence of

the convergent transitions (Proof Obligation (4)) is indicated in the diagram part by the fact that these transitions do not form a loop. The columns "Transitions" and "Refined Transitions" also illustrate partitioning of the error detection events (Proof Obligation (6)). The progress of the refined specification always has to be ensured in line with Proof Obligation (5).

In order to illustrate the idea of progress diagrams in combination with refinement patterns, we use the abstract system (shown in Fig. 3a, Section 4.2.1) consisting of two states (*st1* and *st2*) and two transitions (*tr1* and *tr2*). We refine it to the concrete system shown in Fig. 3b, where the state (*st1*) is partitioned into substates (*st1a* and *st1b*) and the anticipating transition *trNew* is added between the new substates. The progress diagram of this sample refinement step is depicted in Fig. 8.

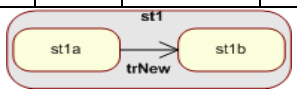
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in state *st1*) – data ref. • adding new transition concerning the substates (*trNew*) – event ref.	*st1*	*st1a* *st1b*	-	*trNew* *(st1a, st1b)*	-
					

Fig. 8. Example of a progress diagram for the pattern 3b from Section 4

In the progress diagram in Fig. 9 we depict the event refinement by separating existing transitions. We continue refining the system shown in Fig. 3b in Section 4.2.1, by detailing its functionality and splitting the existing self-transition *tr1* into self-transitions *tr1a* and *tr1b,* according to the substates separated in the previous step. We also assume that with respect to the added transitions in the refined system we simultaneously add new variables *y* and *z*. The new variables and their initialisation are depicted in the rightmost column of the progress diagram.

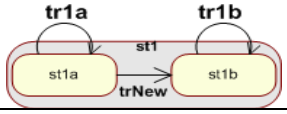
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
2nd refinement step: • separating the existing transition *tr1* into two transitions (*tr1a* and *tr1b*) concerning the substates (*st1a* and *st1b*) – event ref.	-	-	*tr1*	*tr1a* *(st1a, st1a),* *tr1b* *(st1b, st1b)*	*y::* TYPE *z::* TYPE
					

Fig. 9. Example of a progress diagram for the pattern 3c from Section 4

We also consider the flattening pattern to diminish complex hierarchical state structure created while performing consecutive refinement steps. In Fig. 10 we show the progress diagram for the flattening pattern for the diagram in Fig. 3c given in Section 4.2.2.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
3rd refinement step: • removing the superstate (flattening) – data ref.	*st1 (st1a)* *st1 (st1b)*	*fst1a* *fst1b*	-	-	*v*:: TYPE

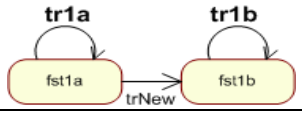


Fig. 10. Example of the progress diagram for the flattening pattern

The progress diagram of the choice path pattern is depicted in Fig.11 and Fig. 12. In Fig. 11 we create a choice path on the transition *tr1*, by refining the transition *tr1* into two more specific transitions *tr1a* and *tr1b*. The guards $G_{1a}(q)$ and $G_{1b}(q)$ are created in such a way that we ensure progress of the system.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
4th refinement step: • adding alternative paths to transitions – event ref. (detailing functionality)	-	-	*tr1*	*tr1a (st1, st2),* *tr1b (st1, st2)*	*q*:: TYPE

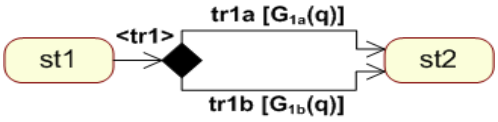


Fig. 11. Example of a progress diagram for the choice paths pattern (specifying functionality)

The splitting could also be used to model more detailed failure. In Fig. 12 we split the transition *tr2undef* between the states *st1* and *susp* into two transitions, *tr2a* and *tr2undef2,* by strengthening the guard condition on the refined transition *tr2a*. The guard of the refined transition *tr1* is the negation of the refined failure transition *tr2a.*

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
5th refinement step: • adding alternative paths to transitions – event ref. (error detection refinement)	-	-	*tr2undef*	*tr2undef2 (st1, susp),* *tr2a (st1, susp)*	*q* :: TYPE

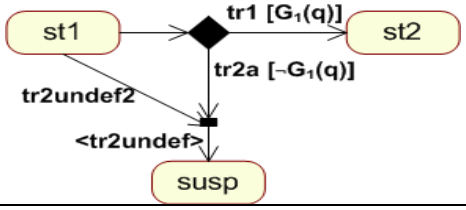


Fig. 12. Example of a progress diagram for the choice paths pattern (specifying error detection)

When adding common behaviour to the existing states (superposition of an orthogonal region), we want to express which of the new transitions are performing the functionality of the old ones. Therefore, we refine existing system (shown in the

upper part of the superstate in Fig. 7) to a functionally more structured and unified one (shown in the lower part of the superstate in Fig. 7). Thereby we create a behavioural pattern, where the system before the refinement is synchronised with the system after the refinement.

In the progress diagram in Fig. 13 we depict the orthogonal region as follows. We show only the lower region without the previous higher region in the superstate. In the tabular part we compare the old superstate with the new one using a bracket notation (superstate {subA1, subA2…} {subB1, subB2…}) to indicate the hierarchy of states in regions. We indicate the states that need synchronisations with existing incoming and outgoing transitions. Furthermore, we specify the new transitions in the orthogonal region and add some variables according to the refinement step.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
6th refinement step: • adding an orthogonal region to the states with common behaviour – data and event ref.	*susp {susp1 susp2 susp3}*	*susp {susp1, susp2, susp3} {unknown(<-fail), known, fixed(->recover), unfixable(->terminate)}*	-	*diagnose (unknown, known), fix (known, fixed), discard (known, unfixable)*	*r::* TYPE

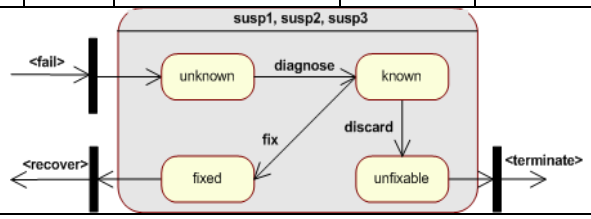


Fig.13. Example of progress diagram for the orthogonal region pattern

6. Case Study Memento

Fig. 14 depicts the progress diagram of the *first refinement step* for the Memento system (following the abstract specification presented in Section 3), where states are partitioned into substates and transitions are added with respect to these. Partitioning the state *init* indicates that the initialisation phase is divided into a connection phase (state *conn*) and a preparation phase (state *prep*), that both need cooperation with the server. The state *susp* is treated in a similar way. Namely, the hierarchical substates *sc*, *sp*, *sr* and *sf* are created, implying that there are, in fact, various ways of handling the errors, corresponding to the states *conn*, *prep*, *ready* and *finalised*. Thereby, more elaborate information about conditions of error occurrence is added. Note that introducing hierarchical substates corresponds not only to a more detailed model in the structural sense, but also in the functional sense. The transitions (events) *tryInit*, *failInit* and *recoverInit* are refined to more detailed ones taking into account the partitioning of the initialisation phase. The self-transition *tryInit* is refined by two events, *tryConn* and *tryPrep*, which remain self-transitions for the states *conn* and *prep,* respectively. The error handling is refined by events: *failConn* and *recoverConn* for the substate *conn*, and *failPrep* and *recoverPrep* for the substate *prep*. The

initiator, transition *cont* (added between the new substates *conn* and *prep*), converges to *tryPrep* and *failPrep* which are refined transitions of *tryInit* and *failInit* respectively. The initiator is guarded by the guard (*is_ok=FALSE*) from *tryInit*, thus ensuring feasibility. New variables *is_conn*, *is_prep* and *wwaited* are introduced to control the system execution flow. Note that there are separate diagram parts (not shown) for the substates *sr* and *sf*.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in states *init* and *susp*) • adding new transitions concerning the substates	*init*	*conn* *prep*	*tryInit* *(init, init)*	*tryConn (conn, conn),* *tryPrep (prep, prep)*	*is_conn*:= FALSE, *is_prep*:= FALSE, *wwaited*:= FALSE
			-	*cont (conn, prep)*	
	susp	*sc, sp,* *sr, sf*	*failInit* *(init, susp)*	*failConn (conn, sc),* *failPrep (prep, sp)*	
			recoverInit *(susp, init)*	*recoverConn (sc, conn),* *recoverPrep (sp, prep)*	

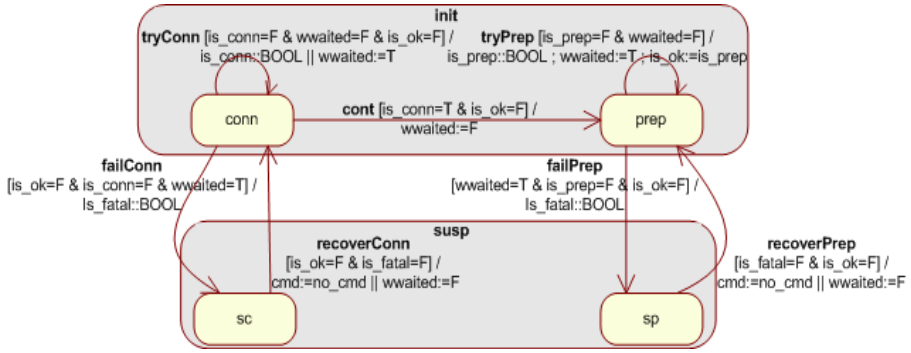


Fig. 14. Progress diagram of the first refinement step of Memento

As the refined specification is translated to Event B for proving its correctness, the progress diagram provides an overview of the proof obligations needed for the refinement step. Since we add new states and variables, we indicate that the old transitions and initialisation need to be refined, according to Proof Obligation (1) and (2). For example in Fig. 14 events *tryConn* and *tryPrep* refine *tryInit*. Event *cont* is a convergent event that only assigns the new variable *wwaited* (Proof Obligation (3)). Since this event is the only newly introduced event in this refinement step and it connects two separate states *conn* and *prep,* Proof Obligation (4) is fulfilled. Furthermore, the error detection event *failinit* is partitioned into *failConn* and *failPrep* in line with Proof Obligation (6). The transitions are composed in such a way that they ensure progress in the diagram (Proof Obligation (5)).

The result of the first refinement step is shown in the statechart diagram in Fig. 15. When comparing this diagram to the one in Fig. 14, it is worth mentioning that even if the former shows the complete system, the diagram is more difficult to read with all its details. The progress diagram shows only the relevant changes in a more legible way.

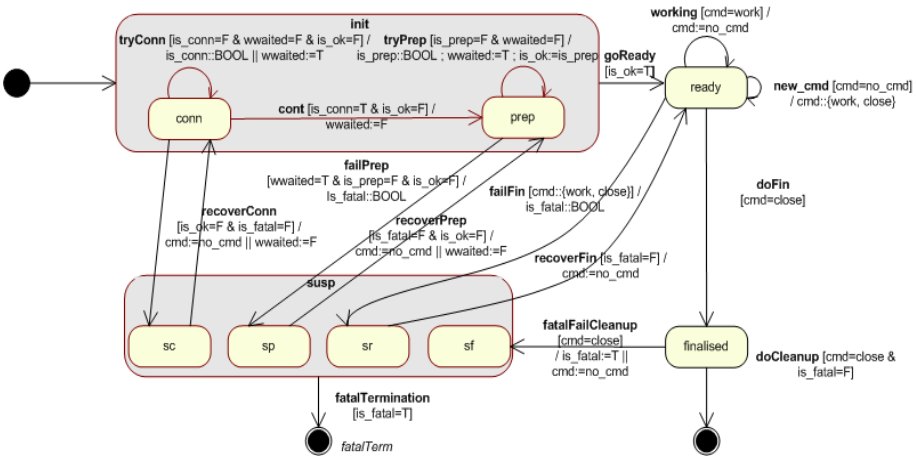


Fig. 15. Statechart diagram of the first refinement step of Memento

The excerpt of Event-B code depicting the first refinement step of Memento system is given below.

REFINEMENT Memento_Ref
REFINES Memento
SEES Data
VARIABLES is_fatal, is_ok, cmd, state, wwaited, is_conn, is_prep, conn, prep, sc, sp
INVARIANT is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
(state=init ⇒ cmd=no_cmd) ∧
init_state ∈ {conn, prep} ∧ susp_state ∈ {sc, sp} ∧
(state=init ∧ init_state=prep ⇒ is_conn=TRUE) ∧
(state=susp ∧ susp_state ∈ {sc,sp} ⇒ cmd=no_cmd) ∧
(state=susp ∧ susp_state=sp ⇒ is_conn=TRUE) ∧
(is_prep=TRUE ⇒ is_conn=TRUE) ∧ ...
INITIALISATION is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
is_conn:=FALSE || is_prep:=FALSE || wwaited:=FALSE || susp_state:=sc ||
init_state:=conn
EVENTS
tryConn (refines tryInit) =
WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ is_conn=FALSE ∧ wwaited=FALSE
THEN is_conn :∈ BOOL || wwaited:=TRUE || is_ok :∈ BOOL **END**;
failConn (refines failInit) =
WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ wwaited=TRUE
THEN state:=susp || susp_state:=sc || is_fatal :∈ BOOL **END**;
recoverConn (refines recoverInit) =
WHEN state=susp ∧ susp_state=sc ∧ is_ok=FALSE ∧ is_fatal=FALSE
THEN state:=init || init_state:=conn || cmd:=no_cmd || wwaited:=FALSE **END**;
tryPrep (refines tryInit) =
WHEN state=init ∧ init_state=prep ∧ is_ok=FALSE ∧ is_prep=FALSE ∧ wwaited=FALSE
THEN is_prep :∈ BOOL; wwaited:=TRUE; is_ok:=is_prep **END**;
failPrep (refines failInit) =
WHEN state=init ∧ init_state=prep ∧ wwaited=FALSE ∧ is_prep=FALSE ∧ is_ok=FALSE
THEN state:=susp || susp_state:=sp || is_fatal :∈ BOOL **END**;
recoverPrep (refines recoverInit) =
WHEN state=susp ∧ susp_state=sp ∧ is_ok=FALSE ∧ is_fatal=FALSE
THEN state:=init || init_state:=prep || cmd:=no_cmd || wwaited:=FALSE **END**;
goReady =
WHEN state=init ∧ is_ok=TRUE ∧ is_conn=TRUE ∧ is_prep=TRUE ∧ init_state=prep
THEN state:=ready **END**;
...
END

In *the second refinement step* new hierarchical substates are added in the state *prep* along with new transitions that make use of them. These hierarchical substates indicate that the preparation phase is actually composed of two phases (program as well as module preparation). This step is similar to the one above and is not further described here.

The third refinement step (Fig. 16) strengthens the guards of the transitions/events (according to the pattern in Fig. 6) and shows a more detailed failure management. New variables, concerning communication with the server, are introduced to express the details of the program preparation phase. These variables represent sending the identification data (*idDataSent*), reading the response (*respRead*), and checking whether the values for response and user are valid (*respValid* and *userValid*). Furthermore, new failure transitions *nIdDS*, *nRR*, *nRV* and *nUV* corresponding to these variables refine the old general failure transition.

Description	States	Ref. States	Transitions	Ref. Trans.	New Var.
3rd step – adding alternative paths to transitions	-	-	*failPrepPr (prepPr, sp)*	*failPrepPr1 (prepPr, sp), nIdDS (prepPr, sp), nRR (prepPr, sp), nRV (prepPr, sp), nUV (prepPr, sp)*	*idDataSent*:=FALSE, *respRead*:=FALSE, *respValid*:=FALSE, *userValid*:=FALSE

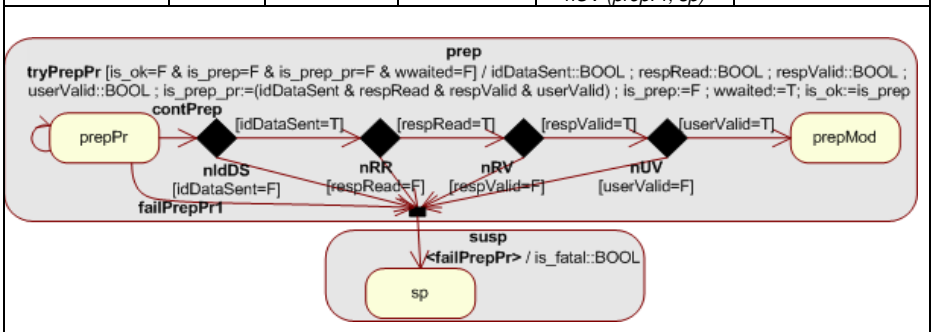


Fig. 16. Third refinement step

Here, the progress diagram also gives an intuitive representation of the proof obligations, now concerning strengthening the guards of the old events (Proof Obligation (2)). This is indicated by the transitions between the *choice point* symbols in the diagram part of the progress diagram. Moreover, the outgoing transitions of these symbols illustrate intuitively that the relative deadlock freeness (Proof Obligation (5)) is preserved. Again the partitioning of the error detection event *failPrepPr* in the columns "Transitions" and "Refined Transitions" visualises Proof Obligation (6).

Below we show the partial Event-B code for the third refinement step of our case study.

REFINEMENT Memento_Ref2
REFINES Memento_Ref1
SEES Data
VARIABLES is_fatal, is_ok, cmd, state, wwaited, is_conn, is_prep, conn, prep, sc, sp, sr, prepPr, prepMod, prep_pr, prep_mod, is_prep_pr, idDataSent, respRead, respValid, userValid
INVARIANT is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧

```
                (state=init ⇒ cmd=no_cmd) ∧
                init_state ∈ {conn, prep} ∧ susp_state ∈ {sc, sp} ∧
                (state=init ∧ init_state=prep ⇒ is_conn=TRUE) ∧
                (state=susp ∧ susp_state :∈ {sc, sp, sr} ⇒ cmd=no_cmd) ∧
                (state=susp ∧ susp_state=sp ⇒ is_conn=TRUE) ∧
                (is_prep=TRUE ⇒ is_conn=TRUE) ∧
                idDataSent ∈ BOOL ∧ respRead ∈ BOOL ∧
                respValid ∈ BOOL ∧ userValid ∈ BOOL ∧ prep_state:=prep_pr ∧
                (state=init ∧ init_state=prep ∧ prep_state=prep_mod ⇒
                    idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ∧
                (state=susp ∧ susp_state=sr ⇒
                    idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ∧
                 (is_prep_pr=TRUE ⇒
                    idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ...
INITIALISATION  is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
                is_conn:=FALSE || is_prep:=FALSE || wwaited:=FALSE || susp_state:=sc ||
                init_state:=conn || prep_state:=prep_pr || is_prep_pr:=FALSE || idDataSent:=FALSE
                || respRead:=FALSE || respValid:=FALSE || userValid:=FALSE
EVENTS
    tryPrepPr (refines tryPrep) =
                WHEN state=init ∧ init_state=prep ∧ is_ok=FALSE ∧
                is_prep=FALSE ∧ wwaited=FALSE ∧ is_prep_pr=FALSE ∧ prep_state=prep_pr
                THEN idDataSent :∈ BOOL; respRead :∈ BOOL; respValid :∈ BOOL; userValid :∈ BOOL;
                IF (idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE)
                THEN is_prep_pr:=TRUE
                ELSE is_prep_pr:=FALSE END;
                is_prep:=FALSE; wwaited:=TRUE; is_ok:=is_prep END;
    failPrepPr (refines failPrep) =
                WHEN state=init ∧ init_state=prep ∧ wwaited=TRUE ∧
                is_prep=FALSE ∧ is_ok=FALSE ∧ is_prep_pr=FALSE ∧ prep_state=prep_pr
                THEN state:=susp || susp_state:=sp || is_fatal :∈ BOOL END;
    recoverPrepPr (refines recoverPrep) =
                WHEN state=susp ∧ susp_state=sp ∧
                is_ok=FALSE ∧ is_fatal=FALSE ∧ is_prep_pr=FALSE
                THEN state:=init || init_state:=prep || prep_state:=prep_pr ||
                cmd:=no_cmd || wwaited:=FALSE || is_ok:=FALSE ||
                idDataSent:=FALSE || respRead:=FALSE || respValid:=FALSE || userValid:=FALSE END;
    nIdDS (refines failPrepPr) =
                WHEN state=init ∧ init_state=prep ∧ prep_state=prep_pr ∧ is_prep=FALSE
                ∧ is_prep_pr=FALSE ∧ is_ok=FALSE ∧ wwaited=TRUE ∧ idDataSent=FALSE
                THEN state=susp || susp_state=sp || is_fatal :∈ BOOL END;
    ...
END
```

The specification presented on the listing above, although more concrete, is not yet implementable. Nonetheless, it provides very good understanding of what actions should be taken in order to ensure stability and fault tolerance.

7. Related Work

Design patterns in UML and B have been studied previously. Chan et al. [6] work on identifying patterns at the specification level, while we are interested in refinement patterns. The refinement approach on design patterns was presented by Ilič et al. [9]. They focused on using design patterns for integrating requirements into the system models via model transformation. This was done with strong support of the Model Driven Architecture methodology, which we do not consider in this paper. Instead we provide an overview of the development from the patterns.

Refinement patterns in the Event-B method were also investigated by Alexei Iliasov [8], but with respect to the rapid development of dependable systems. The author explores a method for mechanised transformation of formal models and merges theory with practice by implementing the tool that supports the formerly created patterns language. Since automation is less error-prone than manual coding, applying patterns with the use of the created tool is profitable for the dependable systems development. We also rely on patterns in order to prevent introducing the errors into the system development, making the construction of the system process more dependable. However, we are not concerned about creating a language for the patterns. Instead we benefit from the progress diagrams through the readability and the intuition they provide.

An approach relating formal and informal development is used in the research of Claudia Pons [17], where the formally defined refinement methodology is submerged into UML-based development. The method is described by the term "formal-to-informal", treated as a complement of the "informal-to-formal" approach standing for translating the graphical notation into formal language. The presented methodology is based on the Object-Z formal language and UML structures. It presents an object decomposition pattern and a non-atomic operation refinement via examples of classes. In our research we focus on statemachines instead of classes and combine the formal and informal approaches, which in our case are complementary to each other. Moreover, we use Event-B in order to have a tool support for our development.

Defining standards in semantics for different level of abstractions in system level design has been studied by Junyu Peng, Samar Abdi, and Daniel Gajski in [15]. The authors' approach to system development relies on the automation of the refinement process via tool support. The main focus in their research is to improve robustness and usefulness of the system design, even if the methodology aims at the architecture of the system in general. Their effort is towards rapid prototyping and evaluation of several design points, while our approach is of a formal nature, focusing on the correctness of the system created in a stepwise manner.

8. Conclusions and Future Work

This paper presents a new approach to documentation of the stepwise refinement of a system. Since the specification for each step becomes more and more complex and a clear overview of the development is lacking, we focus our approach on illustrating the development steps. This kind of documentation is not only helpful for the developers, but also for those that later will try to reuse the exploited features. The documentation is also useful for communicating the development to stakeholders outside of the development team. Thus, a clear and compact form of progress diagrams is appropriate both for industry developers and researchers.

Formal methods and verification techniques are used in the general design of the Memento application to ensure that the development is correct. Our approach uses the B Method as a formal framework and allows us to address modelling at different levels of abstraction. The progress diagrams give an overview of the refinement steps and the needed proofs. Furthermore, the use of progress diagrams during the

incremental construction of large software systems helps to manage their complexity and provides legible and accessible documentation.

In future work we will further explore the link between the progress diagrams and patterns. We will investigate how suitable the progress diagrams are for identifying and differentiating patterns used in the refinement steps. Although progress diagrams already appear to be a viable graphical view of the system development, further experimentation on other case studies is envisaged leading to possible enhancements of the progress diagrams.

We have considered the possibility of developing tool support for drawing progress diagrams and automatically generating a new refinement from the progress diagram and the previous level model. The most likely route for tool support is to extend the UML-B tool [19], which is already an extension of the Event-B tool set. The complete tool set is based on the Eclipse development environment as a 'rich client platform'. UML-B provides a graphical drawing tool for drawing state machine diagrams (as well as class diagrams) and converting them automatically into Event-B where the Event-B static checker and prover automatically perform verification on the model. UML-B uses the 'Eclipse Modelling Framework' (EMF) to generate a repository for UML-B models from a meta-model diagram. The UML-B meta-model defines the abstract syntax of the UML-B language. The drawing tool is based on the 'Graphical Modelling Framework' (GMF). We envisage a new meta-model for progress diagrams, that extends the UML-B meta-model to define the refinement relations, that we have described in this paper, mapping individual elements of an existing UML-B model to newly created UML-B elements. The diagrammatic part of the progress diagram editor would consist of a reduced UML-B Statemachine diagram. The tabular part of the progress diagram is an elegant view of the refinement properties but it is not the most suitable interface for editing them considering that they are based on the existing UML-B meta-classes. Therefore, a new editor interface (diagrammatic, tree structured or tabular) will be developed for defining the refinement properties as extensions to the referenced existing model elements. The tabular part of the progress diagram will be automatically generated as a read-only view of the refinement properties. A builder will be provided to generate the new refined UML-B model based on the progress diagram refinement model. Since the generated model is a UML-B model, the existing tools will automatically generate an equivalent Event-B model as soon as it is created.

Acknowledgements
We would like to thank Dr Linas Laibinis and Dr Dubravka Ilič for the fruitful discussions on the use of the tools supporting the research.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley, 2004.
3. R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In: *Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.
4. R.J.R. Back and K. Sere. From modular systems to action systems. *Software - Concepts and Tools* 17, pp. 26-39, 1996.
5. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language - a Reference Manual*. Addison-Wesley, 1998.
6. E. Chan, K. Robinson and B. Welch. Patterns for B: Bridging Formal and Informal Development. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, LNCS 4355, pp. 125-139, 2007. Springer.
7. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
8. A. Iliasov. Refinement Patterns for Rapid Development of Dependable Systems. *Proc of Engineering Fault Tolerant Systems Workshop* (at ESEC/FSE, Dubrovnik, Croatia), ACM Digital Library, 4 September, 2007.
9. D. Ilič and E. Troubitsyna. A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland, February 2005.
10. S.M. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337-356, April 1993.
11. C. Metayer, J.R. Abrial and L. Voisin. *Event-B Language*, RODIN Deliverable 3.2 (D7), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf (May 2005)
12. Object Management Group. *Unified Modelling Language Specification - Complete UML 1.4 specification*, September 2001.
13. Object Management Group Systems Engineering Domain Special Interest Group (SE DSIG). S. A. Friedenthal and R. Burkhart. *Extending UML™ from Software to Systems*. (accessed 04.05.2007)
14. M. Olszewski and M. Pląska. *Memento system*. http://memento.unforgiven.pl, 2006.
15. J. Peng, S. Abdi and D. Gajski. Automatic Model Refinement for Fast Architecture Exploration. In *Proc of the 15th International Conference on VLSI Design (VLSID.02).*2002, p. 332, IEEE Computer Society .
16. M. Pląska, M Waldén and C. Snook: Documenting the Progress of the System Development. In *Proc. of Workshop on Methods, Models and Tools for Fault Tolerance*, Oxford, UK, July 2007
17. C. Pons. Heuristics on the Definition of UML Refinement Patterns. In *SOFSEM 2006: Theory and Practice of Computer Science*, LNCS 3831, pp. 461-470, 2006. Springer.
18. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92-122, January 2006. ACM Press.
19. C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *Proc. of The IASTED International Conference on Software Engineering – SE2008*, Innsbruck, Austria, February 2008.
20. C. Snook and M. Waldén. Refinement of Statemachines using Event B semantics. In *Proc. of 7th International Conference of B Users (B2007): Formal Specification and Development in B*, Besançon, France, LNCS 4355, January 2007, pp. 171-185. Springer.
21. E. Troubitsyna. *Stepwise Development of Dependable Systems*. Turku Centre for Computer Science, TUCS, Ph.D. thesis No. 29. June 2000.
22. M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design 13(5-35),* 1998. Kluwer Academic Publishers.

Publication 2

Measuring the Progress of a System Development

Marta Olszewska (Pląska) and Marina Waldén

Originally published in:
The book "Dependability and Computer Engineering: Concepts for Software-Intensive Systems", Luigia Petre, Kaisa Sere and Elena Troubitsyna (Eds.), IGI, 2011.

©2012 This paper appears in "Dependability and Computer Engineering: Concepts for Software-Intensive Systems" edited by Luigia Petre, Kaisa Sere and Elena Troubitsyna
Copyright 2012, IGI Global, www.igi-global.com. Posted by permission of the publisher.

Chapter 17
Measuring the Progress of a System Development

Marta (Pląska) Olszewska
Åbo Akademi University, Finland & Turku Centre for Computer Science (TUCS), Finland

Marina Waldén
Åbo Akademi University, Finland & Turku Centre for Computer Science (TUCS), Finland

ABSTRACT

For most of the developers and managers, the structure and the behaviour of software systems represented in a graphical manner is more understandable than a formal specification of a system or than plain code. Our previous work combined the intuitiveness of UML with the development rigour brought by formal methods and created progress diagrams. In progress diagrams, the design decisions within a system refinement chain are assisted by the application of patterns and illustrated in a comprehensible and compact manner. In order to rigorously assess and control the design process, we need to thoroughly monitor it. In this chapter we show how the application of generic refinement patterns is reflected in measurements. We establish measures for the evaluation of the design progress of the system, where the progress diagrams are assessed from the size and structural complexity perspective. Our motivation is to support the system developers and managers in making the design decisions that regard the system construction.

INTRODUCTION

Nowadays software systems are being built using various technologies, appropriate to their application. Project leaders can choose between flexible agile methods (Martin et al., 2001), a spiral methodology (Boehm, 1988) for large and expensive projects, a waterfall model (Royce, 1970) for iterative developments, or prototyping originating from industry, among many others. Moreover, more rigorous approaches, like formal methods (Ladkin & Thomas, 2009), can be used for large, complex and critical system developments, e.g. those intended for space, aircraft or

DOI: 10.4018/978-1-60960-747-0.ch017

Copyright © 2012, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

military sectors. Achieving high quality software is an aspiration of every development (Hayes & Jarvis, 1999). Hence, it is essential for software measurement to be an elementary part of the quality assurance program within every development process, including rigorous ones. When the measurements encompass several metrics, like size, complexity or effort, they can be regarded as reasonable quality indicators.

Since software changes are inevitable, it is crucial to be able to handle them efficiently (DeMarco, 1986). Monitoring the progress of the development benefits in better control and quicker reaction to the changes being made, e.g. by modifying the current modelling approach or refining the model. Given that some measures, like complexity (McCabe & Butler, 1989), can be computed early in the development cycle, they add significant value to managing the software process, as well as the software itself. Moreover, they can contribute to cost reduction and increase the maintainability and modularity.

We believe that the overall quality of the development process, including the systems fundamentals in the design phase strongly influences the quality of the final system. We are interested in modelling large and complex critical systems, embedded systems in particular, and focus on the initial stages of the development. We think that performing the changes at this point is efficient in the sense of time and money. Managing the complexity at this point influences the overall quality of the system and its enhancements. It also impacts specific quality attributes, such as dependability or, more specifically, maintainability or resilience. Therefore, it is possible to diminish the threat against dependability, for example, by constantly controlling the development quality with measurements.

The measurement activities in a software development project should already be applied by the designers. These are the personnel responsible for the modelling decisions, who have to take into account the specificity of the project, knowledge about the application domain and the environmental settings. Also at this point the analyst or quality assurance managers should have access to the collected data in order to present their findings to the project manager, who then makes higher level decisions about the project at its early stage. The testers, as a separate development group, should be aware of the outcome of measurements. The measurement findings can then be juxtaposed with testing results and, as a consequence, testers can propose possible changes.

The combination of well known methodologies that have been proven to be efficient and successful needs to be supported with a good engineering practice. In our research we benefit from formal methods and the Unified Modelling Language, which are merged with measurements for the purpose of quality assurance. A certain quality, including dependability, can be guaranteed and achieved by continuously monitoring the system with measures. The value of the system being created lies in system planning, anticipation of future needs and feasibility of redesign. However, this is not to be accomplished without the innovative use of technologies and continuous supervision of development progress.

One of our goals in this chapter is to show how generic refinement patterns are reflected in measurements. Moreover we want to demonstrate how design decisions influence the measurements and the quality of the system being constructed. Specifically we concentrate on evaluation of progress diagrams and pattern-based development with respect to size and complexity measures. The approach can be extended to tackle a more generic problem of statechart diagrams assessment. Our research results are targeting developers (particularly designers and system architects), as well as project and system managers. The methodology is intended for the design stage of the development and can assist in the system documentation.

This chapter describes a method to tackle complexity in system development using a combination of rigorous approach and graphical notation.

The former allows the developer to accurately identify system functionality, while the latter helps to maintain an insight and comprehension of the complexity and size of the system. The progress diagrams notation that is supported by measurements can be used to detect and eliminate design issues. Moreover it can indirectly aid with aligning the project schedule and budget.

The rest of the chapter is structured as follows. In section *Formal Development* we present an overview of the underlying formal methodology and stepwise refinement of systems, as well as a generic explanation of the Event-B modelling language. The Unified Modelling Language description, with special focus on statechart diagrams in addition to progress diagrams, is given in *Graphical Representation of System Development*. Section *Refinement Patterns* depicts refinement patterns and section *Metrics and Measures* introduces the idea of metrics and measures in concerning size and complexity of a diagram. The application of metrics and measures for certain refinement patterns is analysed, evaluated and partially validated in section *Measures for Refinement Patterns and Progress Diagrams*. The following section is dedicated to a literature review of the state of the art. In the final section some general remarks are given and our future work is presented.

FORMAL DEVELOPMENT

In this chapter we are interested in assessing quality of software for critical systems and, hence, we focus on measurements for lightweight formal system development. This means that we concentrate on a rigorous system development, but do not perform formal proofs. In our work we use Event-B (Abrial, 2010), which is a formal method and a specification language used for system-level modelling and analysis. Event-B is based on Action Systems (Back & Sere, 1996) as well as the B Method (Abrial, 1996), and is related to B Action Systems (Waldén & Sere, 1998). With the Event-B formalism we benefit from tool support for proving the correctness of the development to achieve dependable critical systems via Rodin Platform.

An Event-B specification consists of a *machine* and its *context* that depict the dynamic and the static part of the specification, respectively. The dynamic model defines the state variables and the operations on these. The context, on the other hand, contains the sets and constants of the model with their properties and is referred to the machine through the SEES relationship. All the variables are strongly typed in the invariant, which might also contain properties that should be maintained by the system. The operations on the variables are given as events of the form *WHEN guardTHEN substitutionEND*. *Guard* represents a state predicate, whereas a *substitution* is a B statement expressing how the event influences the program state and is given in the form of deterministic or nondeterministic assignment over the system variables. It needs to be mentioned that the notation for the Event-B language, e.g. symbol ':=' signifying the assignment, is consistently used throughout the chapter.

It is beneficial to construct the system in a stepwise manner in order to be able to efficiently handle the system requirements or implementation details. This gradual introduction of the characteristics of the system to its specification is well supported by the Event-B formalism. The refinement process enables us to transform an abstract specification *A* into a more concrete and deterministic system *C*, which maintains the functionality of *A*. We add new functionality to the system by using new variables and substitutions on these, and relying on the superposition refinement technique. This way a specification preserves the behaviour throughout the whole modelling process and can be proven to be correct by satisfying several proof obligations (Katz, 1993; Snook & Waldén, 2007; Troubitsyna, 2000; Waldén & Sere, 1998). The stepwise development provided

by refinement mechanisms enables managing the development in such a way, that the design decisions are constantly controlled. This results in a well-organised development process and impacts the quality of the final system. It also facilitates the design, as well as inspections and reviews of the model. Furthermore, it gives mathematical rigour to the development.

The systems are constructed gradually, in a stepwise manner, using superposition refinement (Back & Kurki-Suonio, 1983) supported by mechanisms of the tool. Event-B uses set theory as a modelling notation and refinement to characterize systems at all abstraction levels. It also uses mathematical proof to verify consistency between refinement levels. Here we focus on the modelling activity as such, since the proof obligations that provide evidence that the system is correct were thoroughly described in our previous work (Pląska, Waldén & Snook, 2008).

The Rodin Platform offers valuable assistance for modelling, enabling refinement and mathematical proof. It is worth mentioning that since the tool support for the Event-B (http://www.Event-B.org) is developed within the Eclipse Integrated Development Environment (IDE), it is further extendable. Rodin, as a "rich client platform", allows operating on or transforming an Event-B model via several plug-ins. For example, UML-B plug-in (Snook & Butler, 2008) supplies a visual drawing tool, which enables the user to model statemachine and class diagrams and automatically translates them into the Event-B code. Another plug-in that a modeller could use is "finer plug-in" (Iliasov, 2007), which adds the refinement patterns support to the RODIN platform. There are four patterns presented in the latter, all intended for dependable systems. We use the visualisation idea and combine it with the application of refinement patterns. However we concentrate only on parts of the system most affected by refinement and focus on employment of generic refinement patterns.

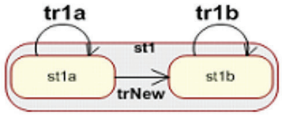
GRAPHICAL REPRESENTATION OF SYSTEM DEVELOPMENT

We believe that (re)inventing a technique for solving a particular group of problems might occur not to be worth the effort, when there are some well-known approaches that can be adjusted to certain needs. Alternatively, using a subset of the features offered by a specific methodology may very well suffice and lead to reasonable results in an efficient and cost effective way. In our case, we combine the adaptation of existing measurement approaches with using a subset of other methodologies for rigorous development of complex critical systems.

We use Unified Modelling Language™ (UML) (Booch, Rumbaugh & Jacobson, 1999; OMG, 2009) as it offers diagrams that are appropriate to our need of modelling the system's structure and behaviour. We benefit from characteristics of statechart diagrams, which are in fact extended versions of statemachines. Statemachines have well defined theoretical foundations, which are inherited by statechart diagrams. They allow designing the dynamic behaviour of a system and at the same time provide good level of abstraction. The system is modelled in such a way that it reacts to certain external or internal stimuli, which activates an action and, ultimately, leads to a change of state.

A statechart diagram basically consists of states and transitions. It clearly and briefly expresses the complex logic relationships between states and the internal behaviour of processes. Hence, the diagram depicts all possible states of the system and the transitions between them. The current state of the system depends on the preceding transition and its associated condition (guard). The state of a system can change in response to certain events (Friedenthal & Burkhart, 2003). Among many of the intricate modelling means, statechart diagrams offer mechanisms for the decomposition (for hierarchical states, also called or-states) and synchronisation (and-states), which enables one to

Figure 1. Example of a progress diagram for separating existing transitions

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
3rd refinement step: • splitting the existing transition *tr1* into two transitions (*tr1a* and *tr1b*) with regard to the substates (*st1a* and *st1b*) – event ref.	-	-	*tr1*	*tr1a* (*st1a*, *st1a*), *tr1b* (*st1b*, *st1b*)	*k*:: TYPE *l*:: TYPE
					

model complex relationships between concurrent states. This is especially useful when developing reactive systems, e.g. modelling parts of embedded systems (Mueller, 2009).

Statechart diagrams are considered useful in all development phases. They allow detecting defects in the design stage by lowering the hypothetical cost of finding the same defect in later stages of the development. Moreover, they can be valuable when finding the incomplete, missing or contradicting requirements. In addition, they can serve as a resource for code generation in the implementation phase or exist for documentation purposes and post mortem analysis.

Since reading and understanding a mathematical notation used in the development of highly critical systems might not seem straightforward to a regular developer, we have supported the Event-B with a visual documentation. In our previous work we have introduced *progress diagrams* (Pląska, et al., 2009), inspired by the idea of UML (OMG, 2009), where software systems are described via diagrams. We believe that carefully merging well-known practices and existing methodologies often occurs to be beneficial and very practical. Hence, we combine formal notation (Event-B) with UML-like diagrams and present it in the form of a table divided into a description part and a diagram part. The description part concisely depicts the relevant features or design patterns of the system in the current development step. Moreover, it shows how states and transitions are refined, as well as represents the new variables that are added according to these characteristics. The diagram gives an auxiliary view and is in fact a part of the statechart diagram which is being refined. We can mention that the notation for the diagrams corresponds to the Event-B notation. Progress diagrams are considered to be useful for communicating the development steps to non-formal methods related practitioners or engineers.

In Figure 1 we show a simple example of a progress diagram, where as a result of event refinement the self-transition *tr1* is being split into self-transitions *tr1a* and *tr1b,* with respect to the substates *st1a* and *st1b*. Moreover, we simultaneously add new variables *k* and *l*, as a consequence of the added transitions in the refined system. In the diagram the symbol '::' denotes nondeterministic assignment.

Progress diagrams assist the construction of large software systems in an incremental and layered fashion. They document the design decisions and show the detailing of the system in successive refinement steps in a compact way. Since they show the part of the system being currently developed, they enable us to focus on the details that we are most interested in, and give a comprehensible picture of the possibly complex system development. The support for system

modelling can be used for design reviews or, when having a tool support, advanced model checking. In the future it may be employed for code and test-case generation. By capturing the essence of changes, progress diagrams help to concentrate on the size and complexity measurements of the most intensively developed parts of the system. Moreover, the modelling supported by patterns could already encapsulate some ready-made measurements.

REFINEMENT PATTERNS

Since we want to have a more controllable refinement process, we apply refinement patterns while designing the system. A pattern is defined as a set of rules depicting how components of an abstract specification are changed to construct a refined version of the specification. We focus on generic patterns and their role in the refinement step. Patterns are thought to be a good engineering practice, since they provide an efficient and disciplined development. They are considered necessary in industry applications, as they add to the well-organised design process and lead to limiting the effort and resources needed for the development. They increase understandability, and therefore maintainability, through reusable artefacts.

Understanding how each pattern influences the complexity of the design allows deducing how certain design decisions impact the complexity of a particular component or system. In our investigation we distinguish several refinement patterns, which we divide into two main categories: data and event refinement. One needs to note that it is common for some of these refinement patterns to be applied in one refinement step. In our research we consider the following generic patterns: basic data and event refinements, flattening of hierarchical states, adding choice paths and superposition of orthogonal regions.

We briefly give the Event-B notation for the patterns (Pląska et al., 2008). A standard event contains a guard G*(V)* and a substitution *S(V)*, where *G(V)* is some predicate on the variables *V,* often given as a conjunction of a number of individual guards, and *S(V)* is some assignment of the variables *V*. The variables *V* are of a general type (TYPE). In all of the pattern diagrams, except for the choice paths pattern, the guards on the transitions are omitted due to legibility.

Data and Event Refinement

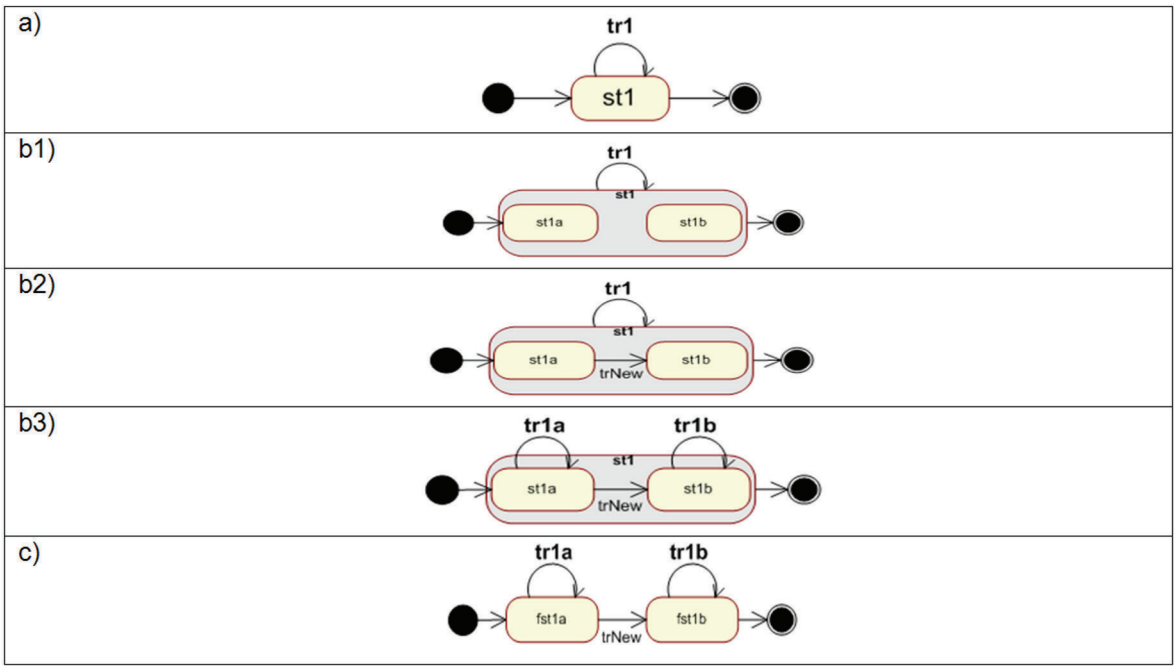
Basic data and event refinements, i.e. splitting the states and adding new transitions, are patterns that are applied most frequently, often at the same time in a single refinement step. In general, the data refinement pattern is employed when a modeller wants to add some more features or variables to the state. Event refinement is used for adding a new transition or separating an existing one. In most cases it is a consequence of data refinement or adding some behavioural characteristics to the system, which is presented in Figure 2, steps b1-b3.

Both refinement types are used in order to reduce non-determinism in the system by strengthening the guards or adding new guards and splitting the existing transitions or adding new ones. All the design decisions should be made in such a manner that they guarantee progress of the system and are in conformance with the proof obligation rules given in (Pląska et al., 2008). An example of an application of the basic data and event refinement pattern is the failure management to model fault tolerant behaviour (Snook & Waldén, 2007).

Flattening of Hierarchical States

Flattening of hierarchical states (or-states) is considered in the refinement strategy as a simple rewriting step, since all the proof obligation rules are trivially preserved. However we consider it

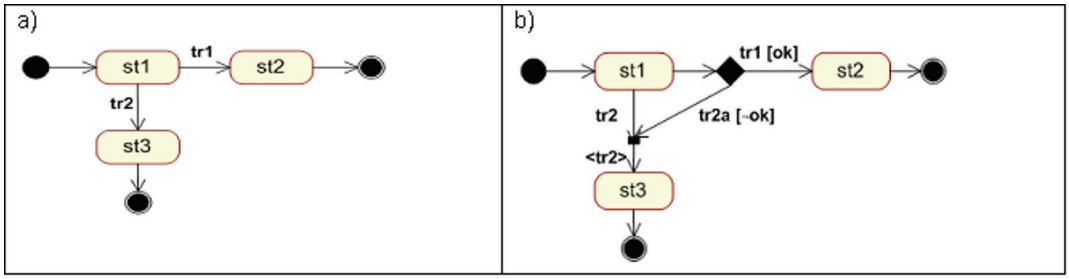
Figure 2. Example of data and event refinement patterns (b1, b2 and b3), as well as flattening of hierarchical states pattern (c)



as a useful way of simplification of the system's hierarchical structure by reducing the level of the nesting. One needs to keep in mind that when modelling the system with superposition refinement and splitting the states in a hierarchical manner, the states are being nested in the superstate. Dealing with the system after several consequent refinement steps might occur to be problematic and at some point hard to comprehend. To make the system more readable and manageable, the flattening pattern is applied. It eliminates the most external superstate and leaves its substates in their original shape. This can be performed under the condition that the superstate to be removed is neither the source nor the target for any of the existing transitions. This means that all the superstate transitions should be moved to the substates, in order for the superstate to become unnecessary. Technically, it involves removing the superstate and create a so called "gluing invariant", which would relate the former model states to the states in the current model. This entails renaming the variables in order to maintain the invariant.

In Figure 2 we present a very simple example showing application of data and event refinement, as well as flattening of hierarchical states patterns. Figure 2a presents the basic statechart containing the initial and final pseudo-states, one simple state *st1* and three transitions, one of them being a self-transition *tr1*. We apply the data refinement pattern and split state *st1* into two or-states *st1a* and *st1b*, obtaining the diagram presented in Fig. 2b1. Next we do event refinement and add a new transition *trNew* between the new states *st1a* and *st1b*, which is portrayed in Fig. 2b2. Then we refine the self-transition *tr1* and split it into two self-transitions *tr1a* and *tr1b*, as depicted in Fig. 2b3. Finally, we apply the flattening pattern and thus reduce the hierarchy level of the statechart by removing the superstate *st1*, which is demonstrated in Figure 2c.

Figure 3. Adding choice paths



Adding Choice Paths

Adding choice paths indicates adding new features to the specification, in order to extend its functionality. It is considered as an event refinement, as the existing transitions are being split and the guards on the events are strengthened. Separating the transition into alternative paths is done using a choice symbol, which is represented by a black diamond symbol. Each choice corresponds to a separate event, where the guard contains the conjunction of all the sections leading up to that path. As a consequence, whenever an event is enabled, a guard permitting it is the conjunction of all the conditions of choice paths leading up to the choice point (Pląska et al., 2008).

In Figure 3 we show the application of adding choice paths pattern. We start from the diagram presented in Figure 3a consisting of an initial and two final pseudo-states, three simple states *st1*, *st2* and *st3* and five transitions, of which two are labelled, *tr1* and *tr2*. We add a choice path and split the transition *tr1* into two transitions *tr1[ok]* and *tr2a[¬ok]*, as shown in Figure 3b. The guards of these new transitions are strengthened and are a conjunction of the section leading to the choice point. By the use of a junction pseudo state, represented by transition marked with angled brackets '<>' that denotes the old, refined transition, (Snook & Waldén, 2007), we show the refinement relation between new transitions and previous abstract ones.

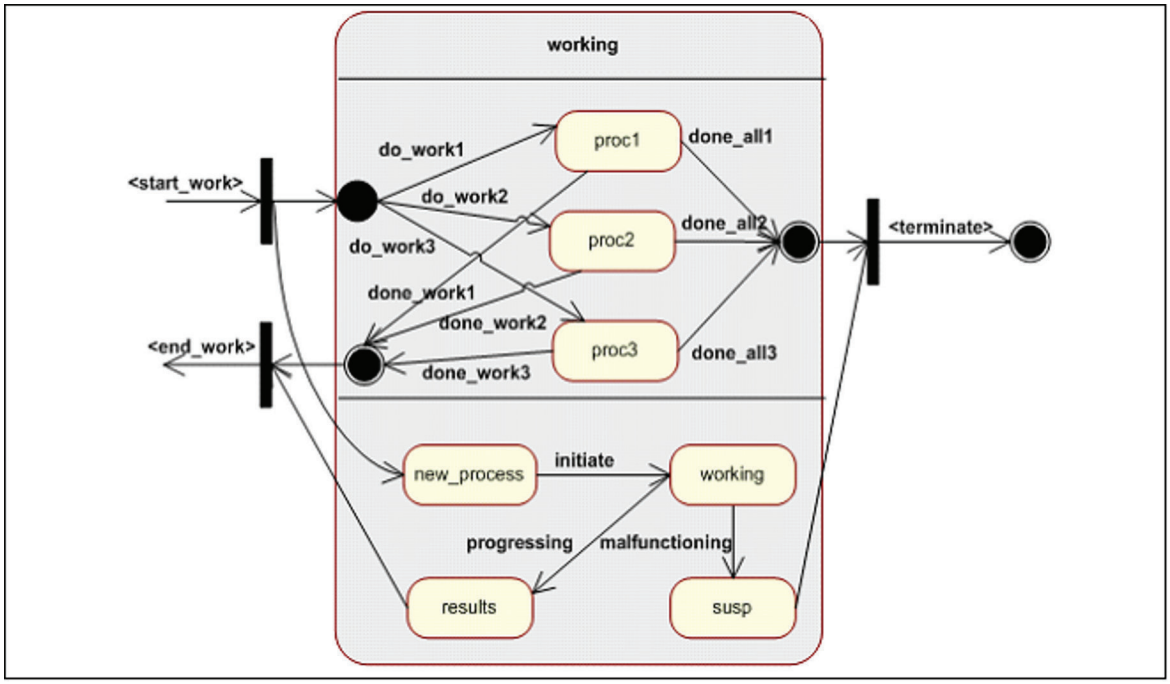
By applying the choice paths pattern we can model failure management, e.g. detailing error detection by splitting the transition denoting undefined types of errors into two new transitions – specific errors, as well as other types of errors.

Superposition of an Orthogonal Region

The orthogonal region, also called an and-state, is a significant element in many practical statechart models. It allows adding the same or similar behaviour to numerous states (Snook & Waldén, 2007). A pattern of adding an orthogonal region is considered as a data and event refinement. It is used in case a number of states have incoming and outgoing transitions (entry and exit transitions, respectively) of comparable functionality. It is applied to reduce architectural redundancy.

By having orthogonal regions, statecharts overcome some limitations of statemachines, i.e. they provide a solution to modelling concurrency. They allow statechart to have substates of a higher level state, which are active at the same time. The substates can communicate with each other according to the following rules. All orthogonal regions of the chart accept events sent to the higher level state. Moreover, one region might create an event as an effect of a transition that is used by another orthogonal region. Finally, it might be tested with a guard whether a different region is in a particular state before allowing a transition to happen to the guarded state (Douglass, 1998). Moreover, guards

Figure 4. Superposition of an orthogonal region



enable conditional event responses, as well as facilitate synchronisation mechanisms.

In Figure 4 we illustrate the application of superposition of an orthogonal region pattern. The system before refinement is shown in the upper part of the superstate *working*. Since we want to add common behaviour to the existing states *proc1*, *proc2* and *proc3*, and thus remove the redundancy, we create a behavioural pattern, which synchronises the systems before and after the refinement. The system after the refinement is given in the lower part of the superstate.

With orthogonal regions modellers can specify extremely complex specification logic in a condensed and comprehensive way. The large state spaces can be decomposed into independent or almost independent parts in an intuitive and natural manner (Harel, 1997). When the number of states increases, the modellers can benefit from powerful statechart characteristics, e.g. orthogonal regions and guards, to gain flexibility in the model and at the same time deal with the complexity issues. Moreover, this solution provides deeper understanding of the underlying model. Since it is based on mathematical foundations, it enables a degree of formalism to be inherent in models created in compliance with rules of orthogonal regions (Meadowcroft, 2005).

METRICS AND MEASURES

Since users expect more characteristics and constantly extended functionality, software systems continuously grow in size and complexity. We want to be able to control the development, not only when the system is being implemented, but already at the early phase of the development. It is cost effective to perform the changes in the system at that point of the creation process (McQuillan, 2006).

Therefore we need to execute measurements with the purpose of analysing and evaluating the early stage of the development. This means that

establishing metrics for the development measurements is necessary in order to monitor and timely react upon the development changes. There exist numerous metrics for certain development methodologies, some of them already quite well established, e.g. O-O programming languages. However, they mostly concentrate on end-product or development process in a perspective of time and human or budget resources.

Nowadays, a requirement for having metrics in projects has shifted from end phases to the early development stages. The collection of measurements, which are intended for the final product, is rich and in many cases quite mature. In contrast, there are either non-existing or experimental measures regarding the specification or design.

This reallocation of focus is caused by the fact that the sooner the data concerning project are collected, the more the development is controlled. This in turn leads to instantaneous reaction upon the problems noticed and more accurate anticipation of challenging issues. The more effective control increases the chance of a project to succeed. Constant control over the project provided by measurements serves as a feedback for the developers. Finally, the statistics can assist managers when making decisions concerning the development and the related resources.

The project managers and quality assurance personnel should use the measurements as means to control the project progress. Moreover, they could benefit by gaining a basis for estimation in the future projects. Constant observation of the collected data concerning the current project enables them to timely identify the factors that are necessary to meet the goal commitments. Here by goal commitments we mean the thresholds set by stakeholders or forced by standards, as well as project requirements. A measurement program should be treated as a part of the development process. A comparison of intended values to the actual ones can aid indication of the weaknesses in the development process. Moreover it can facilitate the process improvement. Managers already benefit from simply having functioning metrics in the project. A lack of measurement program leads to inability of evaluation of the project and thus to diminishing the chances of a product to succeed, which we want to prevent.

There is a need for metrics dedicated for formal development methods, since the current research results are relatively limited, to certain formalisms, like Z language (Hayes & Mahony, 1995; Vinter, Loomes & Kombrot, 1998). In the former publication a static analysis of Z specification notation is performed, whereas in the latter the authors concentrate on linguistic properties of the notation and combine it with predicting erroneous parts of Z specifications. Other sources consider mostly some direct measurements (Whitty, 2002; El Koursi & Mariano, 2002). In our previous research regarding metrics we concentrated on the Event-B formalism and established a complexity metric based on the Event-B syntax (Olszewska (Pląska) & Sere, 2010). The specification language metric concerned the stepwise refinement and its impact on the developed specification.

Here we aim at establishing measures and focus on executing measurements for the design phase, as we think that the data will give us a valuable view at the initial stages of the development. Our goal is to create measures or metrics in such a manner that they will provide guidelines and directions for the early period of the development. We concentrate on establishing structural complexity metric within rigorous developments for progress diagrams that were introduced in (Pląska et al., 2008). We base our measures on the state chart characteristics. There already exist some research on metrics for UML models (McQuillan, 2006), herein statechart diagrams (Miranda, Genero & Piattini), which is, though, mostly based on object-oriented metrics.

The measurement methodology we propose can help the developers to identify the large and complex parts of the design and react on these by reporting them to the managerial staff. The managerial staff should be prepared to aid the

developers with the decision making activities that regard the modelling strategy. The measurements contained in the development process unify the staff involved in the project, since they promote the idea of the shared responsibility with respect to the project goal. At the same time they do not give vague impression regarding the roles of the employees in the project.

Direct Measurements

Measurement is the process of assigning numbers to entities in the real world, so that it is expressed according to some well defined rules (Fenton & Pfleeger, 1997). Direct measurements of an entity involve no other entities for the description. They are taken straightforwardly from the investigated artefact.

We collect direct measurements for progress diagrams based on the syntax of generic UML statechart diagrams. In order to analyse the size of the modelled system and reason about the size changes within the development process we take into account the number of:

- entry and exit actions (*ENA* and *EXA*, respectively)
- states, including simple (*SS*) and composite states (*CS*)
- transitions (*T*)
- guards (*G*)
- events (*E*)

All mentioned metrics, except the number of transitions, produce size measurements. Most of these metrics are self-explanatory. One should note that entry and exit actions are defined as actions that are executed whenever a state is entered and exited, respectively. The simple states also include these simple states that are placed inside the composite ones. Computing the transitions involves summing up the initial and final transitions, self-transitions (identical source and target states) and internal transitions (a source state exists, but there is no target state; no exit or entry activities are executed), as well as generic transitions (distinct source and target states). From the presented list, only the number of transitions and number of simple states can be considered as metrics used for structural complexity calculation later. The theoretical validation of the presented direct metrics was done in (Genero & Miranda, 2003).

In our research we exclude the count of activities, as it is rather straightforward and considered as not essential in our investigation. An activity is an optional behaviour that is executed while being in the state and is represented in a form 'do/ activity'. The execution starts when this state is entered, and stops either by itself or when the state is exited, whichever comes first. The statemachine can model this very behaviour, since a modeller can insert a nested statemachine inside a state, thereby representing an activity.

Collecting the direct measurements can be almost effortless, since it can be automated. Statecharts can be described in the form of e.g. an xml-document or other structured text format. A simple parser or a script should deal with gathering direct statistics from this type of format. There is also a possibility to extend the Rodin platform within the Eclipse environment with a plug-in, which would take a file as an input, parse it and eventually produce direct measurements. The data could then be used as a basis for further analysis or indirect measurement computations, and consequently, producing measurement reports.

Indirect Measurements: Complexity

Establishing the *structural complexity* measure for progress diagrams enables collecting measurements for every development step in the design phase for a certain part of the system. Every modelling decision is reflected in the measurement results we obtain. Whenever the numbers are growing in a fast pace, and let us suppose that they should not, the designer might decide

to change the modelling approach that is used. Complexity here would be employed as a means for numerical analysis of the model at the current development step. It can also be treated in a more "elevated" view, recursively considering all layers in the model hierarchy, that are included in the currently observed one.

In a broader perspective, we recognise complexity as an indicator of a maintainability attribute and refer to it from the point of view of the system modeller. The outcome of complexity assessment should assist not only the developers, but also the managerial staff. Since the success of the project depends largely on the leadership decisions of the managers, the steering party needs evidence of the current state of the project and the related problems. Our metrics facilitate identification of (too) complex parts of the design, and in consequence help managers to decide about changing the development (or design) strategy. The measurement support that functions before the implementation phase can reduce the complexity in earlier stages and prevent the ripple effect of carrying the problems from one phase to another. Immediate fix of the design issues reduces the development costs, which is one of the main managerial goals.

Cyclomatic complexity. We base our complexity model for progress diagrams on the well known and widely used McCabe complexity model (McCabe, 1976). We believe that the definition of new measures by adjusting the existing ones to the current needs can be done efficiently and relatively easy. Shifting the focus from implementation stage (code metrics) to the design stage (model or specification metrics) can help to capture the important development issues well in advance, therefore saving time and money.

We concentrate on the structural complexity measure, which we compute based on progress diagrams. Our purpose is to analyse and evaluate this characteristic and its differences with respect to refinement steps, as well as in perspective of refinement patterns.

Since we focus only on a certain part of the system being modelled with progress diagrams, we can measure and asses this part only. Therefore, making a statement about the complete system that is based only on those results might simply not be possible. Since we show the development steps in sequence, we are able to give the differences between the values of the measures. These can, and in fact are, treated as metric themselves and indicate the characteristics of the modelling progress of the system. Software metrics calculated at the design level and then evaluated can also be used to assess the quality of the development process at its decisive stage.

McCabe has established his *basic complexity model* (McCabe, 1976) in the graph theory, where the computation of this characteristic is based on decision structure of a program, not on the physical size of the system. The general equation presented by McCabe is:

$$v(G) = e - n + 2p,$$

where $v(G)$ is the cyclomatic number, also known as cyclomatic complexity of a graph G, e is number of edges of a graph, n is a number of vertices and p is the number of connected components. For simplicity reasons, p is often assumed to be equal to 1 meaning a single component, therefore giving the basic equation: $v(G) = e - n + 2$.

In (Mills, 1972) it was proven that the complexity of a structured program equals the number of predicates π plus one:

$$CBC = \pi + 1.$$

This approach allows computing *conditions based complexity* (CBC) by straightforwardly counting the number of predicates and thereby removing the need of dealing with the control graph. One can also count conditions instead of predicates.

Since the transitions may include multiple Boolean operators, such as AND or OR, we de-

cided to consider including Boolean operators in the decision count. The cyclomatic complexity with Booleans, also described as strict or *extended cyclomatic complexity* (ECC), is given with an equation:

$$ECC = v_k(G) + NBO,$$

where index k denotes the k^{th} refinement step and *NBO* is the number of Boolean operators. Each time the Boolean appears on a transition, the value of extended cyclomatic complexity increases by one. The interpretation is based on the fact that the conditional branches (transitions) themselves can have an internal complexity, which in sequence impacts the overall complexity. One could model such a conditional statement with several sub-conditions, thereby preserving the complexity level.

Multi-layer structural complexity. Progress diagrams help to visualise the impact of system dependencies, i.e. states and transitions between them, on emerging designs. It is especially the case when the hierarchical dependencies are used. The complexity can be determined for each level of the hierarchy in a rather straightforward manner. However, when the complexity of the complete progress diagram is considered, representing a certain part of the modelled system, it is necessary to recursively cumulate the complexities of the lower-level elements. Intuitively, the more nested the progress diagram, the more complex it is. Hence, it is harder to understand and maintain.

For the progress diagram *M,* without orthogonal regions, its basic *multi-layer structural complexity* measure is computed in the following manner:

$$C(M) = T(M) - SS(M) + 2p(M),$$

where *T* and *SS* are the total number of transitions and simple states in the diagram, respectively. The *p* characteristic denotes the hierarchy level in such a way, that when there is no nesting *p* is equal to one. The presented measure is a simplified equation that includes the OR-decomposition, i.e. hierarchical system design, but leaves out the AND-decomposition.

We validate the structural complexity, and base our examination on the properties defined in (Fenton & Pfleeger, 1997). The structural complexity measure is considered as an internal attribute. It preserves interval scale properties, like preserving order with respect to the attribute, as well as differences between any of the two ordered classes in the range of the mapping. Addition and subtraction operations are allowed. This scale describes information about the size of the intervals that separate the classes. Therefore, it allows understanding the size of the jump from one class to another and assists in registering the changes.

Our measure also fulfils the mathematical properties proposed in (Briand, Morasca & Basili, 1996), such as null value, non-negativity and symmetry. The null value property is fulfilled when the diagram is empty, i.e. there are no states and no transitions. Therefore, the hierarchy level is zero and the overall complexity is also zero. We assume that in a non-empty statechart diagram every state, except the initial state, has an incoming transition, as the system must be able to enter the state. The complexity of the system is non-negative and can be proven by induction. The simplest possible case is when the progress diagram has only one simple state. Thus, the complexity is 1. The reasoning for the non-negativity is presented in detail in section *Measures for Refinement Patterns and Progress Diagrams*. The structural complexity does not depend on the convention selected to represent the relationships between its elements. For that reason the symmetry is preserved.

The above measure did not consider orthogonal states, since the AND-decomposition was excluded. To incorporate the idea of orthogonal states the complexity measure *C(M)* is as follows:

$$C(M) = T(M) - SS(M) + 2p(M) + \sum_{i=1}^{q}\sum_{j=1}^{r_i}\prod_{n=1}^{u_{i,j}} C(M_{i,j,n})$$

where C, M, T, SS and p are defined as in the basic structural complexity equation, q is a number of orthogonal states in M, r_i is a number of abstraction classes in the i^{th} orthogonal state, u_{ij} is a number of orthogonal regions in the j^{th} abstraction class of the i^{th} orthogonal state.

The complexities of the orthogonal states are computed by adding nested sums to the complexity of the diagram. The first layer of the sum iterates over the orthogonal states in progress diagram M. The second sum stands for the number of abstraction classes in the i^{th} orthogonal state. Abstraction classes are defined as orthogonal regions that are fully independent of each other. There is at least one abstraction class for any orthogonal state. The product represents the number of regions u in the j^{th} abstraction class of the i^{th} orthogonal state. M_{ijn} represents the i^{th} orthogonal state of the j^{th} abstraction class and the n^{th} orthogonal region in the considered progress diagram M. The complexity is computed according to the basic structural complexity equation, or recursively, if there are nested orthogonal states.

Note that if no orthogonal regions exist, this formula is reduced to the one presented previously. If the orthogonal regions are completely independent of each other, their combined complexity is additive. This means that the number of independent states needed to model the system is simply the sum of complexities of the regions that the orthogonal state consists of. In other words, it is computed as or-states for each orthogonal region. In the worst case, all orthogonal regions are mutually dependent and communicate with each other by e.g. sending event instances to each other. In this situation the complexity is computed by the multiplication. The validation of the measure is performed recursively, in an analogous manner, as in the case of the basic multi-layer structural complexity equation. The presented formula can also be used for computing structural complexity of statechart diagrams or state machines.

Progress diagrams represent changes made in a single refinement step. Therefore, the complexity of a refinement step can be computed by counting the difference of complexities of two diagrams representing subsequent refinement steps, as given by the equation:

$$C_k(M) = | C(M_k) - C(M_{k-1}) |,$$

where $C_k(M)$ is the complexity of a k^{th} refinement step in the development of system represented by diagram M. $C(M_k)$ and $C(M_{k-1})$ are the complexities of the diagrams M at the k^{th} and the $k\text{-}1^{th}$ refinement steps, respectively.

Comparison of complexity measures. Among all the complexity measures presented in this section, the basic complexity is the original McCabe cyclomatic complexity, which is a generic version used widely in the software engineering society. Conditions based complexity (CBC) is to be used when there is no graph to be employed with the generic McCabe complexity metric; however, one can use this metric directly on e.g. code. Extended cyclomatic complexity (ECC) produces the highest values, which might occur to be disappointing for the software quality assurance experts, but it includes the inner complexities of the transitions. It might also indicate the data intensiveness of the design. Multi-layer complexity is intended for hierarchical systems and fits our refinement modelling profile.

The choice of the particular complexity measure generally depends on personal preferences of the person responsible for measurements. Nevertheless, since we investigate refinement-based systems, which have a tendency to be hierarchical, we recommend using the multi-layer complexity model. For the purpose of additional measurements the extended cyclomatic complexity measure is useful. The reasoning behind this is that the constructed systems are intricate already at the

stage of their low-level structural components, e.g. transitions having multiple conditions (guards).

To give a complete picture of the structural complexity of the considered part of the system, this attribute is computed as recursively aggregated structural complexities of the lower levels in the hierarchy with respect to the general complexity measure. It should be noted that there are certain rules for collecting direct measurements in order to properly compute the structural complexity of a diagram. Those properties are presented with a straightforward analysis and supported by patterns in the following section.

MEASURES FOR REFINEMENT PATTERNS AND PROGRESS DIAGRAMS

We believe that introducing structural and behavioural measures for progress diagrams and rigorously monitoring the development in the design stage will add to the quality of the final system and facilitate complexity handling. It will be possible to guide the development process not only by application of refinement patterns within progress diagrams, but also with measurements. Collected measurements would help the managers to make project decisions concerning organisation of the development process, arranging resources and effective control over the project. Furthermore, they would possibly give feedback to the developers about the modelling approach being deployed. Moreover, there is a need to collect data about the impact of formal approaches on the development process and the final system. This information could be used as evidence for the deployment of formal methods in industry, particularly the sectors constructing dependable software.

We show how patterns relate to measurements and complexity and structure this section as follows: we give a refinement pattern and then examine direct measurements and multi-layer structural complexity for this pattern. We inspect all the patterns given in section *Refinement Patterns* on a case by case basis. Finally, we support the analysis with a simple example and evaluate the approach.

Measures for Refinement Patterns

Data and event refinement patterns influence the complexity in several ways. We first concentrate on the latter. Adding new transitions or splitting existing ones increases the complexity value by the number of transitions added, e.g. when two transitions are added, the complexity increases by two. The placement of transitions in statechart hierarchy does not play any role in the resulting complexity.

As for the data refinement, a state can be added only if there exists at least one transition leading to it. The reasoning is such that introducing a state to which a system cannot enter has no meaning. Adding the state impacts the complexity value depending on where the new state is placed within the hierarchy. There are two cases to consider. If the new state does not increase the level of nesting, the complexity value remains unchanged, provided that the number of the new states is equal to the number of the new transitions added in a single refinement step. It needs to be mentioned that the state(s) cannot be added without adding the transition(s), since the basic data and event refinement are usually done at the same time. The second case to investigate is when adding the new state increases the nesting level, and hereby the complexity.

Flattening of hierarchical states simplifies the statechart diagram and makes it more legible. Although it is considered as a simple rewriting step, it decreases the level of nesting. This in turn, leads to the overall reduction of complexity, which value is lowered at least by two. The measurements follow the general perception of the developers.

Adding a choice path pattern stands for splitting an existing transition into alternative paths. The choice symbol separates the transition in such a

way that the conjunction of the conditions belonging to the new events represents the guards leading to the choice point. The assumptions about the choice paths computation are as follows. The original transition, which is being split, is not included in the total number of transitions count. However, all of the resulting choice paths are incorporated in the direct measurement of transition quantity. This is motivated by the reasoning that the original transition can be in a sense substituted by the conjunction of the choice paths resulting from the refinement step. Therefore, the event refinement in this pattern increases the structural complexity by the number of choice paths minus one, which is in compliance with the common sense.

Superposition of an orthogonal region pattern involves several initial assumptions, before the actual computation of complexity can be performed. It should be emphasised that this pattern introduces concurrency into the statemachine design. Therefore, by intuition, the structural complexity of the model should increase. An orthogonal state is a composite state, where each of its regions consists of simple states. Consequently, the hierarchy level of a diagram with an orthogonal state is always greater or equal to 2. If the orthogonal regions within a state are independent of each other, the complexity of this step increases minimally, since the combined complexity is computed by adding the complexities of each region. In the worst case superposition of an orthogonal region renders the increase in the complexity by taking the product of the complexities of each of its orthogonal regions. This situation occurs when mutual dependency is introduced by establishing communication and synchronisation between the regions. Note that it might be beneficial to apply the superposition of the orthogonal region pattern, which involves an increase in the structural complexity, but removes the redundancy of the design. At an early modelling phase the design redundancy may appear moderately complex. However, the difficulties can be anticipated later on, e.g. at the advanced modelling stage, implementation or maintenance. Redundant modelling solutions can enforce the same type of changes in several places in the design, thus increasing the overall complexity.

Impact of Diagram Constructs on Measures

We discuss elements of statechart diagrams, in order to define how to tackle them when performing structural complexity measurements. First, let us consider deep and shallow history states, which are the pseudo-states used in composite states. They store the information on the configuration of the active states, thereby remembering the recently active substates. Since they are not considered as standard states, they are regarded as supplementary information used for coordinating the configuration of states. At this point of our measurement research, we assume that they do not influence our structural complexity measure. For future work directions we believe that experimentation with deep and shallow history states can lead to including them in the computation of extended cyclomatic complexity (*ECC*), where they would be treated as specific type of conditions.

Other structures, like initial pseudo state, final state, entry and exit point, as well as terminate pseudo state, do not affect our structural complexity measure. They are considered as artificial elements, which are useful for modelling, but do not influence the comprehension of the diagram. Furthermore, the join pseudo state is handled in our theory in the same manner. It is thought to be semantic free and representing the static merging or splitting the existing transitions. The notes, which only provide additional information to the diagram, do not either impact the overall structural complexity.

There are several statechart diagram elements that influence the structural complexity calculation. For example, the junction pseudo-state is used to chain together multiple transitions, each of which can contain a guard. A single junction can have one or more incoming and outgoing transi-

Table 1. Computation of complexity: Example based on the diagrams given in Figures 2-4

Diagram	Type of refinement	*T*	*SS*	*p*	Complexity
Fig. 2a	-	3	1	1	4
Fig. 2b1	data	3	2	2	5
Fig. 2b2	event – new	4	2	2	6
Fig. 2b3	event – splitting	5	2	2	7
Fig. 2c	flattening	5	2	1	5
Fig. 3a	-	5	7	1	4
Fig. 3b	choice paths	7	3	1	6
Fig. 4 (basic)	-	12	3	2	13
Fig. 4 (refined)	orthogonal regions	9, 3, 9	3, 4, 0	1, 1, 2	8+1+13 = 22

tions, each included in the structural complexity computation. Furthermore, join and fork, which represent merging and splitting transitions for orthogonal regions, respectively, impact transition count and affect the overall structural complexity. Although they contain no guards or triggers, they are responsible for synchronisation of concurrent regions or threads.

Example

We give an example of the multi-layer structural complexity calculation based on the diagrams given in section *Refinement Patterns*. The provided example is trivial and it is only supposed to give an intuition on how measurements work in practice.

It is straightforward to observe that the computations executed for the statecharts representing application of the patterns are analogous to the ones done for progress diagrams. The results are given in Table 1, where we describe the diagram for which the refinement is done by providing the number of transitions (*T*), simple states (*SS*), the level of nesting (*p*) and the computed complexity.

We shortly describe how the structural complexity of the multi-layer diagram (Figure 4, refined) is computed. The numbers in the table correspond to, respectively, the upper and lower parts of the orthogonal state and the overall diagram. The upper part consists of 9 transitions and 3 states. Its level of nesting is 1, as all states of this orthogonal region are at the same level. By using our formula we compute the structural complexity of the upper orthogonal region to be $C(M_1) = T(M_1) - SS(M_1) + 2p(M_1) =$

$= 9-3+2*1 = 8$. Similarly we obtain the complexity for the lower orthogonal region which contains 3 transitions and 4 states on the same level. Hence its complexity is $C(M_2) = T(M_2) - SS(M_2) + 2p(M_2) =$

$= 3-4+2*1 = 1$. Subsequently we compute the metric for the overall diagram. There are 9 transitions, mainly due to the use of fork and join. The diagram at this level does not include any simple states. However, the nesting level for the diagram is 2, as it contains an orthogonal state. Since the regions of that state are independent of each other, the complexity of the orthogonal state is a sum of its regions complexity. We use our formula to calculate the final complexity of the diagram to be

$C(M) = T(M) - SS(M) + 2p(M) + (C(M_1) + C(M_2)) = 9-0+2*2+(8+1) = 22.$

Evaluation of the Approach

The presented method is applicable for the systems developed with the use of statechart diagrams, regardless of the domain. The modelling process is extended with measurements, which can sup-

port making the design decisions. The development is being monitored from the early stage and the evidence of certain design decisions is well documented. This is particularly important when it comes to demonstrating the effectiveness of the development process.

When the measurement tool is implemented, it should work in parallel with the modelling process, so that the automatic data collection is performed at the same time. This way it can ensure the constant control over the development and thus facilitate the work of the manager and the quality assurance personnel. Usefulness of the collected data will depend on the interpretation of the results according to the criteria of the project. However no additional roles or competences will be required in the development company in order to fully use the support of the presented measurement methods. At this stage, the main target group to use the measurements for the progress diagrams or statecharts are the developers, in particular the designers.

As soon as the methodology is implemented within the tool and the data collection is automatic, we expect no additional development cost, time and resources necessary to deploy the measurement apparatus. However, the tool that is integrated in the development process can reduce the development cost, e.g. by supporting the developer tackling the design complexity. Our motivation is based on the facts that about 80% of the defects are introduced in the design and coding phases and only half of them is fixed at the phase of origin (according to US National Institute of Standards &Technology). We believe that our methodology facilitates the development in early stages by reducing the number of defects that is introduced, e.g. by complexity issues, in design phase. This in turn reduces the time and effort needed to deal with the design problems in later stages of the development, and thus lowers the project costs.

It is difficult to quantitatively declare how much the proposed method influences software quality and software development progress. For that purpose we would need to either perform a formal experiment or a case study. The former is meant for minor scale and rigorous, controlled investigation, also called research in the small. It involves a lot of preparation and planning, as well as a high level of control in order to provide meaningful results. Therefore, exercising a formal experiment on an industrial project would be rather problematic due to many unmanageable interdependencies that exist in this type of projects. The case study, on the other hand, is constructed as a research in the typical and is suitable for various application domains of significant size. Sister project is a type of case study with certain restrictions regarding the way of proceeding in the investigation. The central factor in the choice is the level of control needed for a formal experiment. In our case executing an investigation with sister projects is more feasible, since we address not only the technical concerns (validating the proposed approach), but also practical matters of research (academic setting, available resources, time and people involved).

The first step states the hypotheses: a null hypothesis and an alternative hypothesis. The null hypothesis expresses that there is no difference in complexity and size between the design created with progress diagrams supported with measurements and a traditional (current) one. The alternative hypothesis says that using the progress diagrams and measurement-supported modelling facilitates the control of model size and complexity. In other words, the control provided by the methodology diminishes the complexity of the design.

In a second step, the state variables are defined: the application area (high criticality: space, military, telecommunications, banking domain), the system type (rigorous systems), the developers experience with the language, tool or methodology

(similar within the group). Then we proceed in the following way: we select two projects, each of which are typical of the application domain, and have the same modelling language and design technique. Furthermore, we take a group of students of similar experience and skills (some experiment or survey considering their abilities should be made in advance). These students will be the persons responsible for the two developments, using the case study provided by some industrial. In the first project they will use some traditional design technique that was used so far. In the second project they will use progress diagrams and measurement-supported modelling. By selecting projects that are as similar as possible we control as much as we potentially can. This will allow attributing any differences in the result to the difference in methodology used. It will be first compared based on the perception of the students. Moreover, some independent expert assessing the ultimate designs may be employed.

The complete validation of the metrics is yet to come. We differentiate four types of threats to validity (Wohlin et al., 2000): conclusion and, construct validity, as well as internal and external validity. Since we have not applied our methodology to many projects, we have no statistical inference to endorse our anticipations. Therefore, we are not able to draw any reasonable conclusions about relationships based on the numerical deduction, which is a threat to conclusion validity. As a consequence, the external validity is only partially achieved. Based on our research examples that we have worked on during the development of our metrics, we can generalise the results only to some extent. The internal validity defined as ability to separate and classify features influencing the examined variables without the researchers awareness has been done when state variables were identified. Furthermore, the construct validity classified as ability to measure the object under study has been done by definition of the hypotheses and investigation of the metrics through our examples. However we are not able to fully conclude our findings, since we lack a number of scientifically significant case studies.

We have been examining our metrics against a number of examples and run it through several special cases. However no advanced or intricate case study has been used, therefore we cannot present any meaningful figures or statistics. The provided example was to visualise the metrics in practise with respect to the refinement patterns presented in the chapter. As a next step we want to apply our method to large and complex (possibly industrial) models and expect it to scale up to handle considerable size and composite designs in an efficient and practical manner. The scalability and practicality of the approach depends on providing the tool support and implementation of the tool.

We are aware about several drawbacks of our method. The main weakness of the approach is the lack of tool support, which could facilitate the data collection and help with the measurement computations. Moreover, some reporting functionality that should be incorporated in the tool can considerably improve the measurement analysis process. Furthermore, application of the methodology to an industrial case study would show the potential of the approach, as well as significantly raise its meaningfulness and value. Only then we can fully verify whether the metrics have the expected effect, or perhaps some fine-tuning of metrics is necessary.

THE STATE OF THE ART: LITERATURE REVIEW

In this section we present the overview of the latest developments in the area of dependable and software intensive systems, as well as related measurement frameworks. It should be treated as

a dedicated literature review consisting of state of the art publications and white chapters, on top of the available documentation of the tools implemented to measure the quality of the design.

Dependability in software has many times been the central topic in the mission- and life- critical areas, such as the military, space or medical domains. It was the industry professionals who indicated the need for methodologies and tools for the creation of dependable software. Threat that a system may fail or malfunction is firmly associated with economic loss. This issue was, and still is, the driving force for research towards ensuring more dependable systems.

There are several frameworks, guides and catalogues that assist in assuring dependable software. A UML-based framework for design and analysis of dependable software, provided by (Kong & Xu, 2008), uses sequence and statechart diagrams to tackle threat-driven modelling and verification of secure software. The methodology is intended for the developers and should reduce the requirement of expertise in formal methods. Our approach also narrows a gap between formal (Event-B) and informal (progress diagrams) modelling, but is much wider, since it involves not only developers, but also managers. Another practical measurement framework (Bartol & Hamilton, 2008) is intended for software assurance and information security at organizational, program or project level. It contains existing measurement methodologies and is meant to aid organizations to integrate software assurance measurement into their projects. Our methodology is finer grained and can be applied at the system design level.

Systems Engineering Leading Indicators Guide (Roedler & Rhodes, 2010) presents measures for evaluating the goodness of systems engineering on a program. It is related to process measurements on the project level and presents the latest results of a cooperation of governmental, industry and academia initiative regarding systems engineering. The proposed objective is to evaluate the suitability of the engineering strategy used in the development of the system. Thus the assessment of the project is done on a high level. We, on the other hand, propose a methodology that focuses on the design phase, which we regard as the most critical in the project.

More general methods for assessment and estimation of software-intensive systems are given in (Stutzke, 2006). Some simple and easy-to-use templates, spreadsheets, and tools are described in order to identify and estimate product size, performance, and quality. In our work we do not consider performance, since we focus on design, not a final product. However we also support the development with product and (partially) process measurements.

Designing and developing real-life software applications, as well as some principles of modern and industrial automation of software design is well described in (Wang & Tan, 2006). This book is dedicated to software engineers and system analysts, who work within the industrial automation domain. Our approach is meant for the same target, but it tackles the safety-critical systems in a more generic manner. It has not been industrially examined yet and still needs tool support in order to establish higher level of automation.

In our work we have already achieved some point of automation due to the use of patterns. This issue has also been addressed by others. For example in the Hillside patterns catalogue (Hillside Group, 2010) a rich directory containing wide variety of patterns is given. Evidently, the use of tools, herein measurement tools, facilitate the development. There are plenty of measurement tools available, just to mention the SDMetrics (SDMetrics) or quality metrics tool for object oriented programming described in (Thirugnanam & Swathi J.N, 2010), both meant for OO designs. There are also more specific, language oriented tools like STAN (stan4j) for Java system analysis. Our design methodology is supported by the Rodin tool. Hence, we want to link our measurement tool support to the Rodin Platform in form of an Eclipse plug-in.

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Nowadays software is present in many disciplines of everyday life. Therefore there are certain requirements that it is expected to fulfil. Among the various qualities that are anticipated dependability is especially important when it comes to critical infrastructures, such as the military or space sectors. Also some standard domains, such as healthcare, transportation, avionics and finance require a certain level of dependability. Software needs to be reliable to a greater extent, as the consequences of its failure can potentially be catastrophic. Therefore, well defined and precise methodologies, such as formal methods, are needed in order to be able to take up the challenges of software-intensive systems.

A rigorous development with tool support for modelling the system is provided by refinement mechanisms within Event-B. By combining it with UML-like structures – progress diagrams – we create a common ground of understanding between mathematicians or computer scientists and system engineers or industrial experts. Graphical representations, such as progress diagrams or statecharts, are also useful when performing design reviews with stakeholders, clients or partners. Progress diagrams give an abstraction of the complex systems and illustrate the significant development changes in a legible and compact manner. Additionally, they document the refinement patterns in more detail. All of this adds value to the deeper comprehension of the refinement decisions and related transformations in the perspective of the collected measurements.

Even though a rigorous development methodology is regarded crucial for obtaining dependable systems, we believe that software in addition needs to be continuously measured throughout the development process. This applies particularly to complex critical and large software systems. Post-mortem analysis, lessons learned, or failure investigations, as in the case of the crash of the maiden flight of the Ariane 5 rocket launcher from the European Space Agency (Lions, 1996) are informative, but also belated. Therefore we apply a lightweight formal development approach in parallel with direct and compound measurements.

Here we have presented the qualitative view of pattern-based rigorous development. The structural complexity metric, which is established and partially validated, is based on the direct diagram measurements. The continuous assessment of the design aids to make timely modelling decisions. The application of the metric is intended for progress diagrams. Nevertheless, it can be extended to measurements of statemachines. We want to evaluate the structural complexity of the stepwise refinement process in order to be able to monitor and control it. Therefore, we perform data collection of direct measurements and compute the complexity according to our equations. This enables us to calculate the complexity of a refinement step, and thus to control the progress of the design.

Future Work Directions

We anticipate that the measurement-supported design leads to a less complex system by reducing the complexity or removing accidental complexity. Moreover, we expect that it reduces the development cost and generally contributes to a higher quality system. However, in order to quantitatively verify that our metrics have the expected effect on the development or end-system we need to perform a formal evaluation and a comparison with some existing metrics.

In our future research we want to establish a threshold for the structural complexity values. This is of high importance, especially when the measurements become an integral part of the development. The development team or project management could use the collected data as guidelines, and thus facilitate the design of the system. Identification of the excessive complexity where it surpasses the quality and usefulness of the sys-

tem would be the main goal of the investigation. The determination would involve studying past projects and comparing the complexity measurement results with expert opinions. When having a hands-on experiment on a reasonable number of projects completed, we could start to use the assembled observations towards a more theoretical approach. We can immediately assume that the threshold will depend on the type of the system and its environment, as well as the experience of the developers and used standards.

Additionally, we consider the development of tool support, which would be implemented as one of the Rodin plug-ins. It would not only enable the drawing of progress diagrams and allow one to instantiate patterns to the previous refinement step, thus creating a new refined model, but also gather the measurements of the model automatically. This would enable monitoring the progress of the design and as a result control the modelling process. Since the creation of overly complex models may lead to changing the modelling approach, the measurements would also facilitate the decisions about the development strategy. Managing the model complexity in the early stage is particularly significant, as it impacts the implementation and influences the maintenance process of the system later on.

Furthermore, we would like to compare the results of our complexity measure for the model with the ones computed for the implementation. Since code generation is one of the objectives that are to be provided as a plug-in for the Event-B, we would like to examine the complexity of the code produced using the basic McCabe model. This would enable us to give feedback on the efficiency and capabilities of the code generators.

REFERENCES

Abrial, J. (1996). *The B-Book: Assigning programs to meanings*. Cambridge, UK: Cambridge University Press. doi:10.1017/CBO9780511624162

Back, R., & Kurki-Suonio, R. (1983). *Decentralization of process nets with centralized control*. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, (pp. 131-142).

Back, R., & Sere, K. (1996). From action systems to modular systems. *Software - Concepts and Tools, 17*, 26-39.

Bartol, N., & Booz Allen, H. (2008). *Practical measurement framework for software assurance and information security*. PSMSC.

Boehm, B. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61–72. doi:10.1109/2.59

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language reference manual*. Reading, MA: Addison-Wesley Professional.

Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, *22*(1), 68–86. doi:10.1109/32.481535

DeMarco, T. (1986). *Controlling software projects: Management, measurement and estimates*. Indianapolis, IN: Prentice Hall PTR.

Douglass, B. P. (1998). *Real-time UML: Developing efficient objects for embedded systems*. Reading, MA: Addison-Wesley.

El Koursi, E., & Mariano, G. (2002). Assessment and certification of safety critical software. In *Proceedings of the 5th Biannual World Automation Congress, vol. 14* (pp. 51-57). Orlando, FL: IEEE.

Event-B.org. (2008). Home of Event-B and the Rodin platform. *Event-B*. Retrieved April 20, 2010 from http://www.event-b.org/index.html

Fenton, N. E., & Pfleeger, S. L. (1997). *Software metrics. A rigorous & practical approach*. Boston, MA: PWS Publishing Company.

Friedenthal, S. A., & Burkhart, R. (2003). *Extending UML™ from software to systems*. Object Management Group, Systems Engineering Domain Special Interest Group (SE DSIG). Retrieved March 31, 2010, from http://syseng.omg.org/Extending%20UML%20From%20Software%20to%20Systems%20-%202003-03-28%20final.htm

Genero, M., & Miranda, D. (2003). Defining metrics for UML diagrams in a methodological way. In A. Jeusfeld, & O. Pastor (Eds.), *Proceedings of ER2003 Workshops ECOMO, IWCMQ, AOIS and XSDM*: LNCS Vol. 2814. *Conceptual modeling for novel application domains* (pp 118-128). Heidelberg, Germany: Springer.

Harel, D. (1997). Executable object modeling with statecharts. *IEEE Computer*, *30*(7), 31–42.

Hayes, I., & Mahony, B. (1995). Using units of measurement in formal specifications. *Formal Aspects of Computing*, *7*, 329–347. doi:10.1007/BF01211077

Hayes, L. J., & Jarvis, A. (1999). *Dare to be excellent: Case studies of software engineering practices that worked*. Upper Saddle River, NJ: Prentice Hall.

Hillside Group. (n.d.). *Design patterns catalogue. Home of the design patterns and host of the PLoP conferences.* Retrieved October 20, 2010 from http://hillside.net/patterns/patterns-catalog

Iliasov, A. (2007). Refinement patterns for rapid development of dependable systems. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems* - EFTS'07. New York, NY: ACM Digital Library.

Jackson, D. (2006). Dependable software by design. *Scientific American*. Retrieved May 01, 2010 from http://www.scientificamerican.com/article.cfm?id=dependable-software-by-de

Jackson, D. (2009). A direct path to dependable software. *Communications of the ACM*, *52*(4), 78–88. doi:10.1145/1498765.1498787

Jackson, D., & Kang, E. (in press). *Separation of concerns for dependable software design*. Workshop on the Future of Software Engineering Research (FoSER). Santa Fe, NM: ACM.

Katz, S. (1993). A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, *15*(2), 337–356. doi:10.1145/169701.169682

Kong, J., & Xu, D. (2008). A UML-based framework for design and analysis of dependable software. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference* - COMPSAC (pp 28-31). Washington, DC: IEEE Computer Society.

Ladkin, P. B., & Thomas, M. (2009, June 22). Formal methods in modern critical-software development. *The Abnormal Distribution*. Retrieved January 4, 2010 from http://www.abnormaldistribution.org/2009/06/22/formal-methods-in-modern-critical-software-development/

Lions, J. (1996). *Ariane 5 - flight 501 failure*. Retrieved May 1, 2010 from http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

Martin, R., Thomas, D., Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., et al. (2001). *Manifesto for agile software development*. Retrieved January 4, 2010 from http://www.agilemanifesto.org/

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *2*(4), 308–320. doi:10.1109/TSE.1976.233837

McCabe, T., & Butler, C. (1989). Design complexity measurement and testing. *Communications of the ACM*, *32*(12), 1415–1425. doi:10.1145/76380.76382

McQuillan, J. A. (2006). On the application of software metrics to UML models. In T. Kühne (Ed.) *Workshops and Symposia at MoDELS 2006*: LNCS Vol. 4364. *Models in software engineering*. (pp. 217-226). Heidelberg, Germany: Spinger-Verlag.

Meadowcroft, B. (2005). A review of statecharts. *Ben Meadowcroft's Web site*. Retrieved March 29, 2010, from http://www.benmeadowcroft.com/reports/statechart/

Merkow, M. S., & Raghavan, L. (2010). Software security for developers. *Networkworld*. Retrieved September 27, 2010.from http://www.networkworld.com/news/2010/092710-software-security-for.html.

Meyer, B. (2006). Dependable software. In Kohlas, W. J., Meyer, B., & Schiper, A. (Eds.), *Dependable systems: Software, computing, networks: LNCS* (*Vol. 4028*, pp. 1–33). Heidelberg, Germany: Springer-Verlag. doi:10.1007/11808107_1

Mills, H. D. (1972). *Mathematical foundations for structured programming. (Report TR FSC72-6012)*. Gaithersburg, MD: IBM Technical.

Miranda, D., Genero, M., & Piattini, M. (2004). Empirical validation of metrics for UML statechart diagrams. In Camp, O., Filipe, J., Hammoudi, S., & Piattini, M. (Eds.), *Enterprise Information Systems V* (pp. 101–108). Netherlands: Kluwer Academic Publishers.

Mueller, P. (2009). State charts can provide you with software quality insurance. *Embedded.com*. Retrieved March 29, 2010, from http://www.embedded.com/design/219400531

Olszewska Pląska, M., & Sere, K. (2010). Towards Event-B specification metrics. In M. Jastram, L. Laibinis, F. Loesch, & M. Mazzara (Eds.), *Proceedings of the first deploy technical workshop* (pp. 81-86). Newcastle, UK: School of Computing Science, Newcastle University.

OMG. (2009). *OMG Unified Modeling Language (OMG UML) infrastructure,* version 2.2. Retrieved October 10, 2010, from http://www.omg.org/technology/documents/profile_catalog.htm

Pląska, M., Waldén, M., & Snook, C. (2008). Documenting the progress of the system development. In Butler, M., Jones, C., Romanovsky, A., & Troubitsyna, E. (Eds.), *Methods, models and tools for fault tolerance: LNCS* (*Vol. 5454*, pp. 251–274). Heidelberg, Germany: Springer-Verlag. doi:10.1007/978-3-642-00867-2_12

Roedler, G., & Rhodes, D. H. (2010). *Systems engineering leading indicators guide*. Boston, MA: Massachusetts Institute of Technology, INCOSE and PSM.

Royce, W. (1987). Managing the development of large software system: Concepts and techniques. In *ICSE: Proceedings of the 9th international conference on Software Engineering* (pp. 328-338). Los Alamitos, CA: IEEE Computer Society Press.

SDMetrics. (n.d.). *SDMetrics - the design quality metrics tool for UML models*. Retrieved October 20, 2010 from http://www.sdmetrics.com/index.html

Snook, C., & Butler, M. (2008). UML-B and Event-B: An integration of languages and tools. In C. Pahl (Ed.) *The IASTED International Conference on Software Engineering* - SE2008 (pp. 336-341). Innsbruck, Austria: ACTA Press.

Snook, C., & Waldén, M. (2007). Refinement of statemachines using Event B semantics. In J. Julliand, & O. Kouchnarenko (Eds.), *7th International Conference of B Users:* LNCS Vol. 4355. *B 2007: Formal specification and development in B*, (pp. 171-185). Heidelberg, Germany: Springer.

STAN. (n.d.). *In stan4j - Structure analysis for Java 2.0*. Retrieved October 20, 2010, from http://stan4j.com

Stutzke, R. D. (2006). *Estimating software-intensive systems: Projects, products, and processes*. Reading, MA: Addison-Wesley Professional.

Thirugnanam, M., & Swathi, J. N. (2010). Quality metrics tool for object oriented programming. *International Journal of Computer Theory and Engineering*, *2*(5), 712–717.

Troubitsyna, E. (2000). *Stepwise development of dependable systems*. PhD Thesis, Turku, Finland: Åbo Akademi University, Turku Centre for Computer Science (TUCS).

Vinter, R., Loomes, M., & Kornbrot, D. (1998). *Applying software metrics to formal specifications: A cognitive approach*. 5th International Software Metrics Symposium, (pp. 216-223). Bethesda, MD: IEEE.

Waldén, M., & Sere, K. (1998). Reasoning about action systems using the B-Method. *Formal Methods in System Design*, *13*(1), 5–35. doi:10.1023/A:1008688421367

Wang, L., & Tan, K. C. (2006). *Modern industrial automation software design: principles and real-world applications*. Hoboken, NJ: John Wiley and Sons.

Whitty, R. (2002). *Research in specification metrics. Colloquium on Software Metrics* (pp. 1–2). IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., & Wesslén, A. (2000). *Experimentation in software engineering: An introduction*. Heidelberg, Germany: Springer.

KEY TERMS AND DEFINITIONS

Measurement: The act or process of measuring in effect of which a figure, extent, or amount is obtained. It is a mapping from the empirical world to the relational world.

Metric: A quantitative measure of the degree to which a system, component, or process possesses a given attribute. A numerical data related to software development or software attribute.

Complexity: A measure of the interactions of the various elements of the software, specifically it is a measure of the design of the software. Complexity measures can be used to give information about reliability and maintainability of software systems.

Analysis: The process of breaking a complex idea into smaller parts to gain a better understanding of it.

Progress Diagrams: A graphically-descriptive notation method, which gives an abstraction of the system modelling process. It illustrates the steps of the development changes within the system refinement and supports the pattern-based development.

Stepwise Development: Development approach, which facilitates the design process by gradually introducing characteristics to the system. Every step implies some design decisions, which allow constructing the system from an abstract perspective to a precise level of detail.

Refinement Patterns: Artefacts that assist system developers in disciplined application of specific mechanisms in system design. They capture successful solutions and enable application to certain recurring problems.

Event-B: A formal method for system-level modelling and analysis. Event-B uses set theory as a modelling notation, refinement to represent model development and mathematical proofs to verify consistency of the models between refinement levels.

UML: Unified Modelling Language (UML) is a standardized general-purpose modelling language created and managed by the Object Management Group. It contains a set of graphical notation techniques that aid the creation of models for software intensive systems.

Publication 3

Specification Metrics for Event-B Developments

Marta Olszewska (Pląska) and Kaisa Sere

Originally published in:
The Proceedings of the CONQUEST 2010, 13th International Conference on Quality Engineering in Software Technology, Dresden, Germany, September 2010.

Based on the publication:
Proceedings of Deploy Technical Workshop, Aix-en-Provence, France, October 2009, Technical Report CS-TR No1187, School of Computing Science Newcastle University, Jan 2010 (Marta Olszewska (Pląska) and Kaisa Sere: "Towards Event-B Specification Metrics")

©2010 International Software Quality Institute (iSQI). Reprinted with kind permission of iSQI.

Specification Metrics for Event-B Developments

Marta (Pląska) Olszewska[1,2], Kaisa Sere[1]

[1] Department of Information Technology
Åbo Akademi University
Joukahaisenkatu 3-5
FIN-20520 Turku
{marta.plaska, kaisa.sere}@abo.fi

[2] TUCS - Turku Centre For Computer Science
Joukahaisenkatu 3-5
FIN-20520 Turku

Abstract

In this paper we define several metrics for Event-B specifications in order to assess the maintainability of a rigorous system in its early development stage. We perform measurements of physical features of the specification. Moreover, we derive metrics like difficulty of constructing a specification or effort needed for its creation. The specification is investigated in the perspective of its syntactical characteristics. We base our metrics on the Event-B language primitives, namely operators and operands that we regard meaningful for our measurement model. We also use statistics about proof obligations in our metrics, as we consider the proving activity as a vital element in the process of creating a specification.

Presented metrics are applied to a number of Event-B specifications, both their dynamic and static parts. They are examined in order to empirically confirm appropriateness for management purposes. Obtained results are analysed in a perspective of an abstract specification and its consecutive refinements.

1 Introduction

Measurements for software systems and their development process are nowadays considered as a good practice in the computer world [Goo04] [AGo02]. They are a part of software projects in order to assure certain quality [KRK05] or for the improvement purposes. They have been evolved for many years, extended for particular development methods like Object-Oriented programming [Lan06] [Mar94] or specialised to meet the needs of certain programming languages, like Java Programming Language [Fra09]. Quality measurements are done not only at the end-product stage, but much earlier, for the requirements [Rob97] or (formal) modelling of the system [Tan08]. An early quality assessment has a major influence on the final product, as a thorough control over the whole development process is maintained.

Research on metrics for formal specifications has already been done for the Z language. In [Hay95] authors perform a static analysis of Z specification notation, whereas in [Vin98] the focus is on the linguistic properties of the notation and predicting erroneous parts of specifications created in Z. An assessment for the B language [ElK02], in

which existing metrics concerned mostly traceability and safety analyses, proof related metrics and direct statistics, like number of LOC (lines of code), variables in a component or imported components, has also been done. However, it has been observed that there is still a need for metrics in the early phase of the development [Whi02].

To the best of our knowledge only direct statistics, like number of LOC, automatic and interactive proofs, invariants, theorems, refinement steps and the like, can be recorded for Event-B in a straightforward way. We believe that providing more elaborated metrics will facilitate a thorough specification analysis and assessment. Furthermore, it will support the improvement of the modelling strategy and possibly enable effort prediction. We anticipate the primary users of our metrics to be managers, who can monitor the project in its initial stage. Obtained information will be a benchmark of how the current development approach influences the specification and, ideally, allow gathering experience on improving the process of creating such specifications. We expect that metrics will be present already when creating a specification and assist developers with modelling and analysis later on.

Event-B is a formal method and a specification language, which is used for system-level modelling and analysis [Eve10]. An Event-B specification consists of a *machine* and its *context* that depict the dynamic and the static part of the specification, respectively [Abr96]. The language is supported by a tool called Rodin Platform [Rod10], which is an Integrated Development Environment based on Eclipse framework. Since the Rodin Platform is a "rich client platform", it enabled us to extend the core with measurement plug-in for automated data collection.

In our work we focus on the syntactical properties of the specification. We benefit from the existing metrics and incorporate them to our early stage development measu rements. e deriv e from alstead' s metrics [Hal77], which describe a program as a collection of tokens that can be classified as either operators or operands. These metrics are used to determine a quantitative measure, e.g. program length, vocabulary size or program volume. Empirical studies show that standard Halstead metric is a good indicator for program length, provided that the data needed for the equations are available [Kan03]. This means that the program, or in our case specification, should be (almost) completed, which seems to be a downside of this metric that we are aware of. However, it can be useful in gathering the information about a size of a specification during the modelling process. It should also be mentioned that we are aware of the strong debates and criticism on the methodology and the derivations of equations [Ham82]. In our research we adapt the general ideas of Halsted to a more abstract, higher-level setting, i.e. formal specifications. Moreover we do not use any estimates, instead we focus on adjusting concrete metrics by including factors specific for formal development. To our knowledge, no experimentation with Halstead metrics for specifications has been done yet.

We use the objectives of the Halstead model and carefully adjust them to Event-B language specifics. Our motivation is that the Event-B syntax is appropriate to be investigated in terms of Halstead metrics, as it contains all the primitives needed for further

computations. These measures should be considered for use during the modelling as outliners of the complexity and development trends. Moreover, the generic version of Halstead metrics, although criticised, is simple and well known to the computer society, as it has been used for almost forty years. We believe that experimenting with the syntactical metrics originating from a programming language will be successful in Event-B case and the results of this investigation will be meaningful.

Our paper is structured as follows. In Section 2, we describe the Event-B language for machine and context. Section 3 depicts the reasoning of the syntax based measurements and presents our concept of metrics for Event-B specifications. In Section 4 we shortly discuss the validity of the given metrics. Section 5 illustrates our research with a large real-life example and analyses the measurement results. We conclude and present plans for the future work directions in Section 6.

2 Event-B characteristics

Event-B [Abr961] is an extension of the B Method formalism [Abr96] and is used for modelling of event-based systems. An Event-B development incorporates the Action Systems formalism [Bac90] [BSe96], which allow the system to be constructed gradually and taking into account refinement rules. Stepwise refinement helps to tackle all the implementation issues by decomposing the problems to be specified and introducing details of the system to the specification in a stepwise manner. In the refinement process an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible.

A specification of a system consists of behavioural part, described by a machine, whereas the data structures are given in a context. The machine model defines the state variables, as well as the operations on these. The context, on the other hand, contains the sets and constants of the model with their properties and is accessed by the machine through the SEES relationship [Abr96].

An Event-B machine consists of its *name*, list of distinct variables *var*, the invariants *Inv(var)*, and a collection of events evt_i, including INITIALISATION, and is depicted with Fig.1.

MACHINE *Machine_0*
SEES *Context_0*
VARIABLES *var*
INVARIANTS *Inv(var)*
INITIALISATION *Init*
EVENTS
evt_1
…
evt_N
END

Fig. 1: General form of machine in Event-B specification

All the variables *var* are declared in the VARIABLES clause and initiated in the INITIALISATION clause. Furthermore, they are strongly typed by constraining predicates of invariants given in the INVARIANT, which might also contain properties that should be maintained by the system. The operations on the variables are given as events of the form ***WHEN*** *guard* ***THEN*** *substitution* ***END***. In case the event is parameterised it is given as ***ANY*** *witness* ***WHERE*** *guard* ***THEN*** *substitution* ***END***, where *witness* is a local variable visible within an event. *Guard* represents a state predicate, whereas a *substitution* is a B statement describing how the event affects the program state and is given in the form of deterministic or nondeterministic assignment over the system variables [Use07].

Event-B classifies events as 'convergent' if they are new events that are expected to eventually relinquish control to an old (refined) event (i.e. it must decrease the variant). Events that are not convergent are classified as 'ordinary' (default). A third classification, 'anticipated', refers to events that will be shown to be convergent in a future refinement. Events can also be categorised as 'extended' or not. In our research we consider all of the classifications, however the 'extended' type is only partially supported with metrics, e.g. when a machine includes it among other types of events.

Context part of Event-B specification is associated with machine by **SEES** relationship. A context is made of its *name*, list of distinct carrier sets *sets*, list of distinct constants *const*, and list of its properties given by axioms *axm* and theorems *th*, which is exemplified in Fig.2. The sets and constants of an abstract context are kept in its refinement, via 'extend' mechanism, therefore they are accumulated.

CONTEXT *Context_1*
EXTENDS *Context_0*
SETS *sets*
CONSTANTS *const*
AXIOMS *axm*
THEOREMS *th*

Fig. 2: General form of context in Event-B specification

3 Metrics for Event-B specification

We derive our metrics for Event-B machines from the Halstead model [Hal77]. The model is based on a collection of tokens, operators and operands, where four primitive measures can be distinguished:

- n_2 - number of distinct operators in a program,
- n_2 - number of distinct operands in a program,
- N_1 – number of operator occurrences,
- N_2 - number of operand occurrences.

These measures are a foundation of Halstead model, which consists of equations expressing the vocabulary, the overall program length, the potential minimum volume for

an algorithm and the difficulty level that indicates software complexity. Moreover, features like the development effort can be specified. One should keep in mind that the accuracy and the behaviour of this model varied between its application domains.

Our tool for automatic data collection deals with the variables, invariants and event blocks of a current machine, and collects the data for each of the machines separately without recognising the semantic relations between the machines. This means that the 'extended' type of event refinement is not considered for the machines, yet. The refinement and extension method is fully supported for the context part of the Event-B specification. Contexts are handled accumulatively, i.e. we acknowledge that considered context is an extension of the previous one. Therefore, the data is collected recursively, with respect to the occurrence of certain set or variable in current context and its presence in previous one. Although we realise that refinement has an impact on the qualitative and quantitative aspects of the specification, currently we consider it in a limited way. Additional issues concerning the metrics acknowledging refinement are given in Section 5 as a part of our future work.

In order to be able to adjust the Halstead model to Event-B environment we have to make several assumptions considering the primitives. Firstly, we decide upon the meaningfulness of operators [Jor99] [Use07] both for the machine and context. As an operator we consider unary operators, binary operators except functions, range operator (symbol **..**), forward composition (symbol **;**), parallel product (symbol ||) and direct product (symbol ⊗). We also consider quantifiers, except separators for set comprehension and bounded qualification (symbols | and **.** respectively), to be operators in our model. The full list of language operators can be found in the language description [Met05]. We gather the data about the number of distinct operators (n_1), as well as the number of their occurrences (N_1).

Secondly, we determine the operands, which we chose to be witnesses and variables [Met05]. In the machine part we count the number of unique operands (n_2) and the number of their appearances (N_2) respecting their visibility rules. If a witness name is declared in several events in a single machine, each of such occurrences is distinct due to the scope. Witnesses are visible only inside a single event, whereas variables are global for the machine. For the context part of the specification we consider sets and constants as operands. We depict counting of the operators and operands with an excerpt of a simple Event-B code [Abr10] presented in Listing 1 and 2.

Listing 1. Example of Event-B machine

```
MACHINE part_1
REFINES part_0
SEES    part_ctx
VARIABLES j, h, l
INVARIANTS
     inv1   :      j ∈ 0‥n
     inv2   :      h ∈ 1‥n ↣ ℕ
     inv3   :      l ∈ 0‥j
     inv4   :      ran(h) = ran(f)
     inv5   :      ∀m·m∈1‥l ⇒ h(m)≤x
     inv6   :      ∀m·m∈l+1‥j ⇒ x<h(m)
```

```
EVENTS

     INITIALISATION   ≙
BEGIN
        act3   :      j ≔ 0
        act4   :      h ≔ f
        act5   :      l ≔ 0
END

     progress 2   ≙
WHEN
        grd1   :      j≠n
        grd2   :      h(j+1)≤x
        grd3   :      l=j
THEN
        act1   :      l ≔ l+1
        act2   :      j ≔ j+1
END

     progress_3   ≙
WHEN
        grd1   :      j≠n
        grd2   :      h(j+1)≤x
        grd3   :      l≠j
THEN
        act1   :      l ≔ l+1
        act2   :      j ≔ j+1
        act3   :      h ≔ h ⩤ {l+1 ↦ h(j+1)} ⩤ {j+1 ↦ h(l+1)}
END
END
```

Listing 2. Example of Event-B context

```
CONTEXT
     part ctx
CONSTANTS
     n
     f
     x
AXIOMS
     axm1   :      n ∈ ℕ
     axm2   :      f ∈ 1‥n ⇸ ℕ
     axm3   :      x ∈ ℕ
END
```

In Listing 1 we show an extract of the Event-B machine, whereas in Listing 2 we present the code of the context seen by this machine. The machine given in Listing 1 contains 8 unique operators and 25 operator occurrences, as well as 3 distinct operands and 42 operand occurrences. Sample context given in Listing 2 consists of 1 unique operator, which is repeated 3 times, in addition to 3 operands that appear in the code 7 times in total.

Having defined primitive measures, we identify several metrics for machine and context parts of a specification and list them after their description. It is worth mentioning that these metrics do not depend on text formatting, like in a case of using LOC as a specification size metric. They are more credible, as the primitive measures are clearly defined. For comparison, in case of LOC it might not be obvious whether to include comments or data definitions to the size computations [Fen97] [FeN00].

We define the size of a *vocabulary* (*n)* of a specification (machine or context) is defined as a sum of distinct operators and operands (1). A sum of operator and operand occurrences (2) is a *size* of a specification (*N)*. Next metric, *volume* (*V)*, represents the information contents of the program. The calculation of *V* is based on the number of operations and operands present in the machine (3).

1. $n = n_1 + n_2$
2. $N = N_1 + N_2$
3. $V = N * log_2(n)$

We have done numerous experiments by applying this metrics to Event-B specifications. The results confirmed a strong correlation between the size metrics, explicitly the vocabulary size *n* and length *N* of each machine. This is a direct consequence of the *n* and *N* definitions, which implies that *n* is always less or equal than *N*. There is also a correlation between *volume* and the machine size metrics (*N*, *n*, LOC), which is particularly visible when the logarithmic scale is used in the diagrams. Therefore *V* could be used as a meaningful size metric, as long as there exist at least one operator or operand ($n>0$). Size metrics are proportional to the direct Event-B machine statistics, such as the number of events or the number of actions. Moreover, they are proportional to the number of constants and axioms in a context.

Another metric, *difficulty level D* (4), representing the difficulty experienced during writing a specification, is proportional to the number of distinct operators n_1 and occurrences of operands N_2, and inversely proportional to the number of distinct operands n_2.

4. $D = (n_1/2) * (N_2/n_2)$

One should note that in practice, since there is a possibility that no operators are used in a machine (empty events with skip) or in a context (either no axioms or theorems are present or sets and constants are given without their properties), the result of D after computation could be undefined.

Construction of a specification is influenced by the problem that needs to be modelled, the clarity and completeness of requirements, the skill of the modeller, as well as the number of interactive and automatic proof obligations (IPO and PO). The difficulty model, which we presented, is suitable for Event-B specifications, as it correlates with the number of IPOs, or in case there are no IPO, with the number of POs.

We also experimented with the Halstead effort general equation ($E=V*D$, where *V* and *D* stand for volume and difficulty, respectively), straightforwardly applying it to Event-B environment, but no regularity or pattern could be observed. Therefore we approached the effort characteristic from the point of view of practitioners. Constructing a specification consists of several factors, such as modelling and proving activities, and is influenced not only by quality of requirements, but also by the power of proving mechanisms and experience of the modeller (also in the interactive proving tasks). We define effort as dependant on the number of proof obligations (automatic and interactive), the machine's volume *V* and difficulty *D* (5).

5. $E = (1+IPO/PO) * V * D$

In the equation the proving factor $(1+IPO/PO)$ is placed to deal with a case when all proof obligations are discharged automatically. Two special cases must be considered. When there are no proof obligations, the proving ratio is treated as zero and the formula (5) is reduced to $E=V*D$. If D is equal to zero or is not a number, the formula (5) should be computed without the difficulty factor. It is worth mentioning that although the Rodin tool might not display any POs, there always exist some trivial POs generated and proved by the platform provers. Such cases do not involve any proving effort. However, the modelling effort is still regarded in the metric. One needs to observe that the lower the value of E, the easier and simpler is to change and improve a specification or maintain it.

The effort characteristic has no established threshold, yet. However, it can be used by developers as a source of information for comparison of how much effort is necessary when using different modelling approaches for a given problem. It can also assist when building the experience of the developers on modelling. It will also help to approximate the effort needed to tackle problems similar to the ones that have already been dealt with. As it is computed in parallel with the specification creation process, it can be checked and decided if it is worth to maintain the chosen modelling solution.

4 Discussion on the metrics validity

Here we concisely apply the Evaluation Framework presented in [Kan04] to our metrics. The main purpose of our measurement is to assist in a self-assessment and improvement of the development process in its early stage. One of our goals is to provide a method for evaluation of the Event-B specifications to the developers and managers. The metrics are intended for the design stage of the development and can assist in the analysis and quantitative assessment of specifications with respect to refinement mechanisms. They are intended to be applied throughout the system modelling process.

Our generic objective is to determine the size of an Event-B specification, the difficulty of modelling it, as well as the effort considering its construction and proving. The variability of these metrics is inherent from the direct measurements that we automatically collect, i.e. number of operators and operands, in addition to their occurrences. The tool that we use to count the direct measurements determines the natural variability of readings and reduces the possibility of measurement errors. As for the foreseeable side effect of using our metric instrument, we consider the lack of existing thresholds as the biggest drawback.

We also validate our metrics against several mathematical properties, such as the non-negativity, null value and module additivity. The size measures given in this paper, such as size of the vocabulary n, length N, as well as the direct measurements considering the unique operators and operands and their occurrences have been validated and proved already in [Bri96] as correct size measures. They are measurable on a ratio scale, as they preserve ordering, size of intervals and ratios between entities. They have a zero

element that represents the total lack of attribute and is a starting element in the measurement. Moreover, arithmetic operations can be meaningfully applied. The volume *V* does not fulfil the additivity of disjoint modules with respect to [Bri96]. Assuming that V=0 for an empty specification, we consider the metric as a ratio scale measurement.

The difficulty level *D* is an ordinal scale measurement, since its values can only be compared according to the ranking it represents. The same applies to the effort metric *E*, since it contains the measurements of ordinal scale type. Both metrics fulfil the non-negativity property.

5 Experimental setup and results

Our methodology was applied to a number of specifications created within the European Project DEPLOY. The specifications are modelling subsets of a highly critical, large-scale and complex industrial system within the space sector. We performed a single domain experiment, with the same type of the development and the same workforce. This made the investigation more credible, as we reduced the number of experiment variables that could skew the overall outcome. The results given by the metrics correspond to the perception of the practitioners.

Here we present a comparative study and an evaluation of our metrics applied to an abstract specification and its subsequent refinements. This enabled us to quantitatively assess the progress of the development from the point of view of the physical characteristics of the constructed specification. Fig. 3 and Fig. 4 present the measurements, which are obtained from one of the software specifications provided by DEPLOY's industrial partner, for the machines and contexts, respectively.

In Fig. 3 we can observe that the displayed statistics correlate in most of the machines. The lack of the line for IPO in machines M0, M1, M2 and M5 denotes that the value of IPO is zero, meaning that all of the proofs have been discharged automatically. It cannot be shown on the diagram because of the logarithmic scale used in the figure.

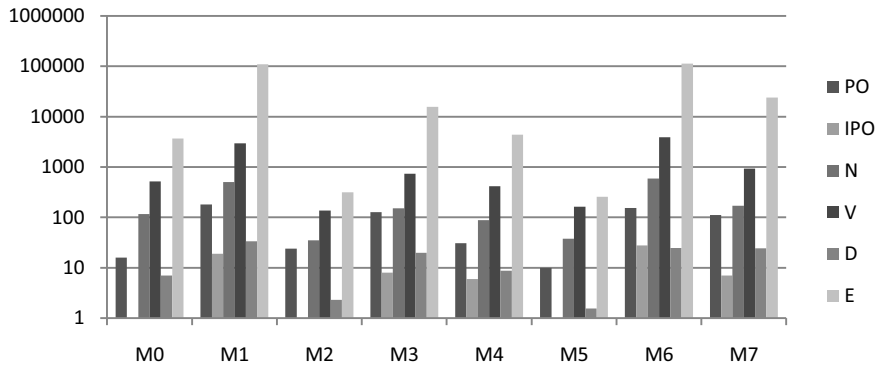


Fig. 3. Statistics on consecutive machines

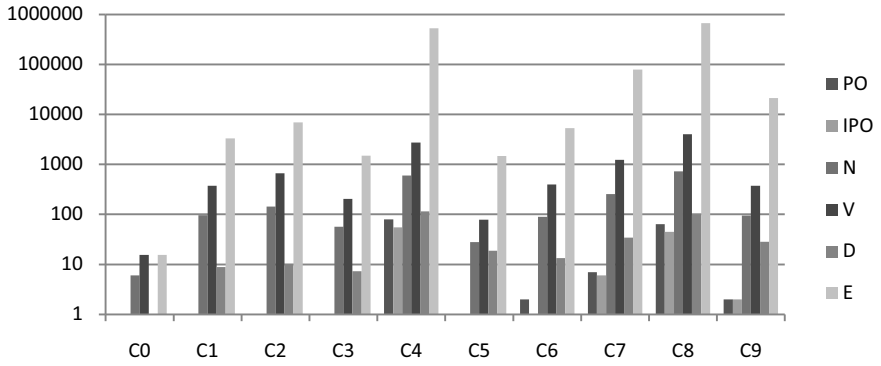


Fig. 4. Measurements on successive contexts

The size measurements N and V correlate in each refinement step and are proportional to well defined number of LOC (non empty and non comment) or other direct size measures, like previously used size measure – number of specification pages. This analysis was done in order to validate our results against understanding of specification size by developers.

The difficulty level D correlates with the number of IPO, and when there are no interactive proves, it correlates with the number of automatically generated proof obligations PO. Consequently, it confirms the perception that the difficulty in creating a machine depends not only on its size (by definition), but also on the proving factor. The effort E is a composite of proving, difficulty level and the information contents of the machine, which is visible as correlation of the listed features in the diagram.

Similar analysis can be performed for Fig. 4 presenting the measurements for consecutive contexts of examined specification. As with Fig. 3, the value for missing data is zero, which cannot be represented on a logarithmic scale used in the diagram.

Having the measurements for each of the artefacts at every refinement step we are able to reason about the progress of the development at the early stages. In case the calculated numbers are increasing excessively at certain step, the developers might decide to change their modelling approach and decompose their problem into several smaller ones. This way the product metrics relate to improving the process of system specification.

We are interested in probing the presented metrics for Event-B specifications in different domains, e.g. military or transportation. We will pursue the search for relations between our metrics and other indicators, e.g. effort results and actual man-hours required to create a specification.

5 Conclusions and future work directions

Nowadays (software) systems keep growing in size and complexity. Therefore, moni-

toring complexity already at the beginning of the project benefits not only the design process as such, but also impacts later development phases, e.g. programming or system maintenance. There is a need for metrics, which would assist in controlling complexity at the early stages of the development.

In this paper we described a quantitative approach to assess the dynamic and static parts of Event-B specifications. The proposed measures are defined within the measurement scales and mathematically validated. Moreover, they have been discussed with the practitioners and, as a result, confirmed the perception of the developers. Our metrics for Event-B provide a better understanding of the physical features of specifications. They facilitate an assessment of an early-stage development. By analysing an abstract specification and its consecutive refinements, they support the modelling process. The presented metrics can be used as a basis for further measurements, like predictions of time and human resources necessary for a development. Therefore they can be treated as an early development indicator for managerial purposes.

We consider refinement mechanism as a vital part of the system design. Our metrics for the context part of a specification with 'extend' mechanism already cover the refinement issue. The data for the measurements are collected and computed accumulatively with respect to the previous refinement steps. However, there is a need to expand our metrics considering machines. For now we focus on the machine refinement, where the events are not being an extension of the previous refinement step events. Therefore each machine is handled separately in our measurement computations. We need to elaborate our approach to be able to fully support the 'extended' type of event refinement in machines. In this situation we will be able to analyse the accumulated Event-B machines with respect to all refinement types throughout the modelling process.

We are aware that presented measures may not suffice by themselves as specification metrics and that there is still a need for metrics that encompass the semantic relations within a specification. However, we believe that results of our research are a good starting point for the further investigation in the field of specification assessment.

As a continuation of our investigation we plan to examine the presented metrics on specifications from other critical domains. Our metrics can naturally be customised to fully reflect the intuition of the developers in other fields. Moreover, we intend to create a set of global metrics, which will consider a complete Event-B refinement process, i.e. the machines and the related contexts for each refinement step.

Acknowledgements

This work has been done within the EC FP7 Integrated Project *Deploy* (grant no. 214158). Authors would like to thank Dr Linas Laibinis and Anton Tarasyuk for the valuable insight on the Event-B language and modelling, and Mikołaj Olszewski for his programming involvement.

References

[Abr10] Example by J-R. Abrial, http://valhalla.cs.abo.fi/~mplaska/QuickSort.zip

[Abr96] Abrial J-R, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press (1996).

[Abr961] Abrial J-R, *Extending B without Changing it (for Developing Distributed Systems)*, Proceesings of 1st Conference on the B Method. Springer-Verlag, Nantes (1996)

[AGo02] Gopal A, Krishnan MS, Mukhopadhyay T, Goldenson DR, *Measurement Programs in Software Development: Determinants of Success*, IEEE Transactions on Software Engineering, Vol. 28, No. 9, (2002).

[Bac90] Back RJR, *Refinement Calculus, Part II: Parallel and reactive programs. Stepwise Refinement of Distributed Systems*. Springer-Verlag (1990), pp.67-93.

[Bri96] Briand Lionel, Morasca Sandro, Basili Victor, *Property-Based Software Engineering Measurement*, IEEE Transactions on Software Engineering, (1996), pp.68-86.

[BSe96] Back RJR, Sere K, *Superposition refinement of reactive systems*, Formal Aspects of Computing, 8(3) (1996), pp.1-23.

[ElK02] El Koursi EM, Mariano G, *Assessment and certification of safety critical software*, Proceedings of the 5th Biannual World Automation Congress. IEEE, Orlando (2002)

[Eve10] Event-B.org, http://www.event-b.org/index.html

[FeN00] Fenton Norman, Neil Martin, *Software metrics: roadmap*, Conference on the Future of Software Engineering. Limerick, Ireland (2000)

[Fen97] Fenton NE, Pflegger SL, *Software Metrics. A Rigorous and Practical Approach.*. PWS Publishing Company (1997).

[Fra09] Metrics 1.3.6, http://metrics.sourceforge.net/

[Goo04] Goodman P, *Software Metrics: Best Practices for Successful IT Management*. Rothstein Associates Inc. (2004).

[Hal77] Halstead MH, *Elements of Software Science*. Elsevier North Holland (1977), pp.128.

[Ham82] Hamer PG, Frewin GD, *M. H. Halstead's Software Science - A Critical Examination*, Proceedings, 6th International Conference on Software Engineering, ICSE. IEEE, Tokyo (1982)

[Hay95] Hayes IJ, Mahony BE, *Using Units of Measurement in Formal Specifications*, Formal Aspects of Computing, 7 (1995), pp.329-347.

[Jor99] Jorgensen M, *Software Quality Measurement*, Advances in Engineering Software, 30 (1999), pp.907-912.

[Kan03] Kan SH, *Metrics and Models in Software Quality Engineering*. Addison-Wesley (2003).

[Kan04] Kaner Cem, Bond Walter, *Software Engineering Metrics: What Do They Measure and How Do We Know?*, 10th International Software Metrics Symposium. IEEE Computer Society Press, Chicago (2004)

[KRK05] Kandt RK, *Software Engineering Quality Practices*. Auerbach Publications (2005).

[Lan06] Lanza M, Marinescu R, Ducasse S, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer (2006).

[Mar94] Martin RC, *OO Design Quality Metrics. An Analysis of Dependencies.*, (1994).

[Met05] Metayer C, Abrial J-R, Voisin L, *Event-B Language, RODIN Deliverable 3.2 (D7)*. (2005)

[Rob97] Robertson J, Robertson S, *Requirements: Made to Measure*, American Programmer, X (1997).

[Rod10] Rodin Platform, http://www.event-b.org/platform.html

[Tan08] Tang A, Tran MH, Han J, van Vliet H, *Design Reasoning Improves Software Design Quality*, Quality of Software Architectures. Models and Architectures. Springer, Heidelberg (2008)

[Use07] *User Manual of the RODIN Platform, Version 2.3*. (2007).

[Vin98] Vinter R, Loomes M, Kornbrot D, *Applying Software Metrics to Formal Specifications: A Cognitive Approach*, IEEE International Symposium on Software Metrics, (1998), pp.216.

[Whi02] Whitty RW, *Research in Specification Methods*, IEE Colloquium on Software Metrics, (2002).

Publication 4

Quality Analysis of Simulink Models

Marta Olszewska (Pląska), Mikko Huova, Marina Waldén, Kaisa Sere, Matti Linjama

Originally published in:
The Proceedings of the CONQUEST 2009, 12th International Conference on Quality Engineering in Software Technology, Nuremberg, Germany, September 2009. dpunkt.verlag GmbH Heidelberg, Germany.

Earlier version:
Marta Pląska and Marina Waldén. Quality Comparison and Evaluation of Digital Hydraulic Control Systems, TUCS Technical Report number 857, Turku (Finland), December 2007.

Related papers:
Pontus Boström, Marta Pląska, Mikko Huova, Matti Linjama, Mikko Heikkilä, Kaisa Sere and Marina Waldén. Contract-based Design in Controller Development and its Evaluation. In NODES 09: NOrdic workshop and doctoral symposium on DEpendability and Security, Linköping, Sweden, April 27, 2009.

Mikko Huova, Marta Pląska, Lauri Siivonen, Matti Linjama, Marina Waldén, Matti Vilenius and Kaisa Sere. Controller Design of Digital Hydraulic Flow Control Valve. In Proceedings of 11th Scandinavian International Conference on Fluid Power (SICFP'09), Jun 2009.

Pontus Boström, Mikko Huova, Marta Olszewska (Pląska), Matti Linjama, Mikko Heikkilä, Kaisa Sere and Marina Waldén. Development of Controllers Using Simulink and Contract-Based Design. In the book "Dependability and Computer Engineering: Concepts for Software-Intensive Systems", L. Petre, K. Sere, and E. Troubitsyna (Eds.), IGI Publishing House, July 2011.

©2009 dpunkt.verlag GmbH Heidelberg. Reprinted with kind permission of dpunkt.verlag.

Quality Analysis of Simulink Models

Marta Pląska[1], Mikko Huova[2], Marina Waldén[1], Kaisa Sere[1], Matti Linjama[2]

[1]Department of Information Technology
Åbo Akademi University
Joukahaisenkatu 3-5
FIN-20520 Turku
{marta.plaska, marina.walden, kaisa.sere}@abo.fi

[2]Department of Intelligent Hydraulics and Automation
Tampere University of Technology
P.O. Box 589
FIN-33101 Tampere
{mikko.huova, matti.linjama}@tut.fi

Abstract

In this paper we present a measurement framework for complex control systems developed in Simulink environment. Quality analysis of a digital hydraulics control system is done for the purpose of its evaluation and improvement. We concentrate on the assessment of quality of the development process and the created software in perspective of lightweight formal methods.

1 Introduction

Continuous software improvement is observable everywhere nowadays. It is also a major goal for most software organizations. The evaluation of quality of produced software is considered as a main activity towards achieving high quality products. Our goal for performing a measurement program is to monitor and evaluate the quality of control system software and its development process.

Most of the developers have some intuition about the quality of software they produced [Jor99]. However, to genuinely determine the quality and obtain its complete picture, measurement activities and appropriate metrics are necessary. The analysis, which takes place after data collection and gathering measurement results, is the final outcome of the examination. It portrays the quality features of the system, by assigning values to the tested characteristics in question, thus giving them certain meaning. Measuring software quality is very complex, because it consists of evaluating software products, processes and resources, each of which is composite as well. Assembling these aspects of quality allows us to get a comprehensive view on the developed system.

Assessing the quality of control software systems, like the digital hydraulics ones, is challenging, as it is multidimensional and algorithmically complex. Digital hydraulic systems are an alternative in the fluid power technology, replacing high-cost analogue valves with simple and easy to manufacture on/off-type valves. Digital hydraulic valves

consist of digital flow control units (DFCU) composed of parallel connected on/off-valves [LKV03]. With this kind of systems it is possible to achieve more flexible functions than with traditional ones. These functions include e.g. energy saving capabilities due to distributed nature of the valve system [MLi07].

To realize these sophisticated functions with just on/off-valves the control algorithms have become more complex. The difficulty with complex controllers is that if they are not properly designed and produced the risk of malfunction becomes higher. To be able to handle the complexity and produce reliable software Boström et al. [Bos07] proposes the use of contract-based design method in producing control algorithms for modern hydraulic systems. In this paper we focus on the test application development, later called TC II, which is a part of controller of digital hydraulic valve system that controls one cylinder. TC II was developed with use of contract-based design methodology in the MATLAB/Simulink environment [Mat07], a high-level graphical design environment for modelling, simulation, implementation and testing of dynamic and embedded systems [Sim09]. The data for the quality measurements were gathered simultaneously within the development of TC II. Therefore we not only can examine the product quality including its structure, but also we are able to observe and analyse the process of the controller development. We also depict project quality including resources, tools and methodology used.

One should note that the term “measurements” can be used with different meaning both in hydraulics and software (system) field. Software related definition states about assigning a number or symbol to an entity in order to characterise software quality attribute [Fen97]. Since software for digital hydraulics is nowadays crucial for obtaining effective and efficient system, a need for software measurements arose. Numerous books, just to mention [Kan03] [KRK05], and publications, for instance [Fen00], [Gra94], were published on quality of software engineering. They discuss the relevance of quality measurement and its contribution towards improving quality of software, as well as introduce software metrics. Structural measurements have been used for managing quality in terms of complexity in [LMB09]. However, we use a broader complexity model for the quality assessment in specific environment (Simulink).

In section 2 we describe the application domain with short history of digital hydraulics. Section 3 specifies the TC II that we later on examine. The contract-based design development methodology, an essential factor influencing the quality of the final system, is depicted in Section 4. The quality metrics and the corresponding measurement results of our study are given in Sections 5 and 6, respectively. In Section 7 we conclude and give directions for our future work.

2 Application Domain – Digital Hydraulics

The need for software development methods with digital hydraulics can be understood when considering the advances of the control algorithms. Many benefits can be gained using digital valves instead of analogue servo and proportional valves. If simple On/Off-

valves are manufactured in large series, the cost of the hydraulic valves may be reduced. Because the response time of the DFCU is the same as the response time of the single on/off-valve, the former is not dependent on the amplitude [LiV07]. Fault tolerance of a digital hydraulic system is much better than fault tolerance of a traditional hydraulic system, because fault in a single valve does not paralyze the system [Sii05]. Different ways to improve energy efficiency of the hydraulic system are also possible with digital hydraulics [Lin08].

Figure 1 presents cylinder control with four digital flow control units. A system of four DFCUs controlling cylinder is known as a digital valve system. This kind of distributed valve system is used to independently control flow rate from supply line *P* to cylinder chambers *A* and *B*, and from the cylinder chambers into the tank line *T*. A typical system consists of four DFCUs that have five parallel on/off-valves, resulting in 20 valves in total. Because each valve can be individually controlled and has two possible states, the number of different control solutions is 2^{20}, resulting in over one million possibilities to choose from. To be able to select the optimal valve control solution at each time step, a model based control algorithm is used.

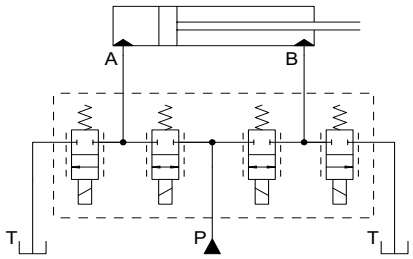


Fig. 1: Hydraulic diagram of digital hydraulic valve system.

The control algorithm consists of four main components. The first component chooses the right control mode according to the load force that is acting on the cylinder, velocity reference and the pressure of the supply line. The control mode describes which of the DFCUs should be used to direct the main part of the flow to achieve optimal efficiency. The second part of the control algorithm is used to select the most promising control solution candidates to more accurate and computationally demanding analysis. Selected candidates are fed to the steady-state calculation component, which calculates the pressures in cylinder chambers and piston velocity that would result from the use of each control candidate. The last component in the control algorithm uses the cost function to select the optimal control signals to the valves using the calculated pressure and velocity information.

Reliability of the control algorithm is very important in heavily loaded hydraulic systems. Incorrect control of the valves may seriously damage the system itself or cause injuries to people who use the system in the worst case. A proper design method has to be used, in order to be able to produce reliable control software. The use of design by contract in development of MATLAB/Simulink based controllers of digital hydraulic

systems has been suggested [Bos07]. In order to study the suitability of contract-based design approach in a development process of such software, TC II was designed.

3 TC II (Test Application)

The cost function of the digital hydraulic valve controller was chosen as TC II (subsystem *Cost Function and Optimal Control*). The task of the cost function is to choose the optimal control signals for the on/off-valves. The cost function is used to simultaneously enable the control of different features of the system. The cost function has to be able to find a compromise between pressure and velocity tracking. Other aspects that the cost function should be able to handle are minimisation of power losses and minimisation of unnecessary valve switching.

At the beginning of the design process the desired properties of TC II were considered at an abstract level. The cost function should be flexible enough to be used with different valve configurations and the system should be as easy to use as possible. The tuning parameters should be intuitive and the user interface well documented. The controller should also give user information about the chosen control signals. Other desired properties of the cost function were reliability and expandability to meet future requirements.

The Simulink diagram with the basic structure of the subsystem Cost Function and Optimal Control is shown in Figure 2. The algorithm is divided into five subsystems, all of which have clear area of responsibility. The subsystem *Velocity terms* is used to calculate the velocity error and cost term for each valve control combination in the search space. The subsystem *Pressure terms* calculates the pressure error for both cylinder chambers and the corresponding cost term. The *Switching terms* calculates three different types of cost terms about the switchings of the valves. The cost terms concerning energy consumption and the opening of the secondary DFCUs are calculated in the subsystem *Secondary DFCU terms*. The fifth subsystem uses calculated cost terms to sum up the value of cost function for each control signal candidate in the search space. The selection of the final valve control signal is also done in the subsystem *Finding the optimal u* by selecting a control candidate with the lowest cost function value.

4 Contract-based design

When considering large and complex control systems, formal methods are recognised to be the approach that effectively manages their reliability. Stepwise development based on *superposition refinement* [BKS83] [BaW98] [Kat93] [BaS96] allows the system to be modelled in a layered fashion at different levels of abstraction. The methodology is present from the very beginning of a development process. It not only assists when specifying the system from the requirements, but also facilitates the development in later stages - design and modelling phases in addition to testing.

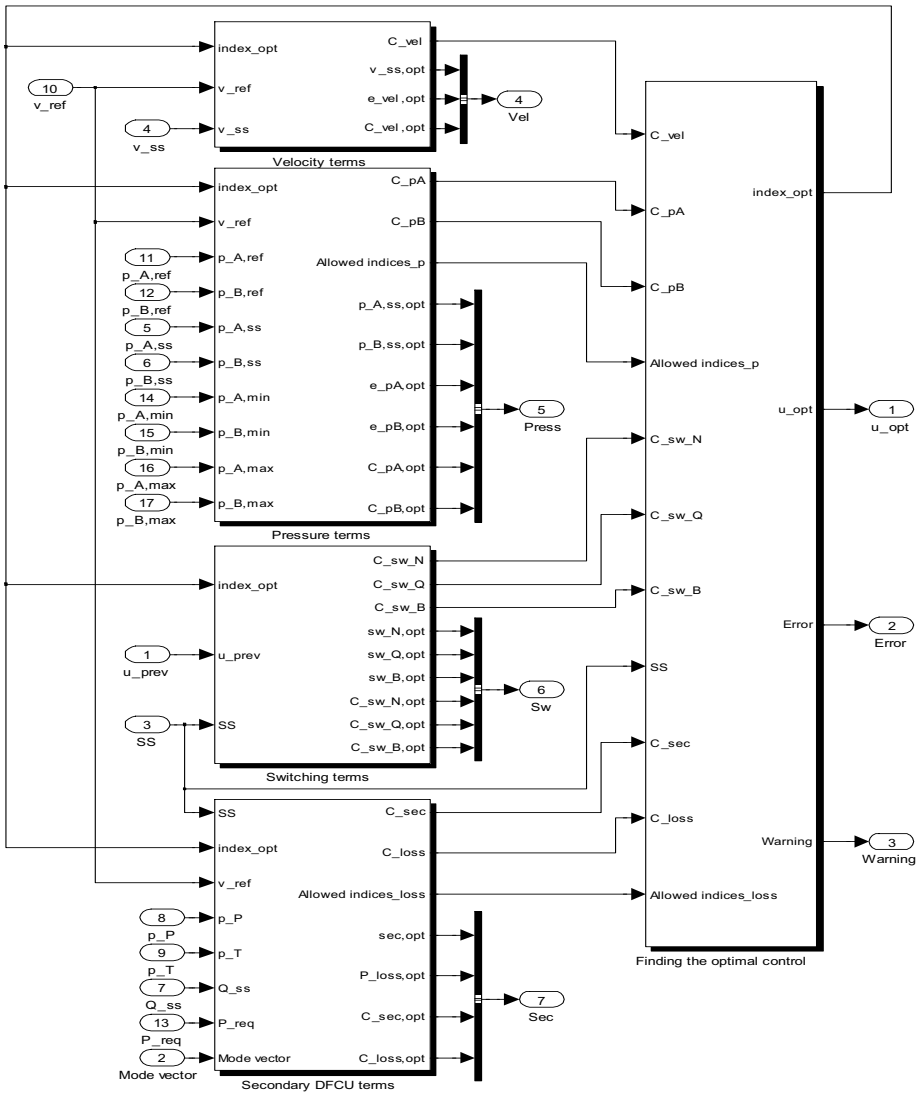


Fig. 2: Structure of the cost function (subsystem Cost Function and Optimal Control).

It is crucial to be able to reason about the elements of the model and their interaction. Therefore we benefit from *Contract-based design* method [Mey92] [Bos08], a systematic approach to specify and implement software. The developed system is seen as a set of communicating components, in our case subsystems. Connections between them are based on accurately defined specifications of mutual require-ensure contracts. Input condition is introduced in the *require clause* (pre-condition) and output condition is introduced by *ensure clause* (post-condition). The specification is associated with every software element. Specifications (or contracts) manage the interaction of a given

element with the real-life elements in terms of obligations and benefits. Every element in the contact is supposed to accept some obligations (pre-conditions), and produce some benefits in return (post-conditions). These characteristics apply to individual routines. For the properties which hold for all instances of classes, class invariants are used. They are assertions of special type and describe the constraints that apply to subsequent changes and cover mutual consistency for the elements. All these properties are examples of assertions or logical conditions related with elements in the contract. Contracts increase the reliability of the developed systems [Bos07].

In the development of TC II we used lightweight formal method, in which proofs were omitted in favour of testing and simulation, as main analysis techniques. The abstract specification was thoroughly prepared with the use of mathematical notation or detailed description. It explained the real world requirements and its dependencies on a high level and was refined into more detailed, lower level specification. Later stages of the development, design and coding, as well as testing, were performed with respect to this specification, ensuring the reliability of the system. This stepwise development guaranteed preserving system's functionality and behaviour given in the initial specification. The whole development process was thus made more manageable and the developers' decisions were more justifiable.

Contracts have been used when modelling TC II in *Simulink*, as they are helpful in reasoning about the development of the model [Bos08]. They ensure that the tasks are performed in a correct manner with respect to the specification and provide robustness, as they assist with handling abnormal situations delivering exception handling mechanisms (errors, warnings as Outport blocks). The notions of inheritance, polymorphism and redefinition are well supported in contract-based design theory. Therefore, it is suitable for stepwise system development, where in each step the system becomes more detailed and clearly defined. In Simulink environment this translates to layering of subsystem blocks and detailing their functionality. The principle of subcontracting ("require no more, provide no less") in the contact methodology is appropriate in this type of systems as well. The assertion-based method facilitates systems analysis due to its precise notation, which prevents risky and premature implementation commitment. In addition, it is mainly used for modelling, as the internal details are out of the scope. It supports software design and project management, as it focuses the developers' and managers' attention on the most significant issues of the development. Furthermore, it is useful in standard documentation of the software elements, as well as communication between developers. This in turn increases the understanding of the system being developed and results in better maintainability of the system later on. The controllers in digital hydraulic systems are becoming more complex and computationally extensive, due to, for instance, many tuning parameters. The similar tendency can also be observed in more traditional machine automation development. Therefore, systematic methods are essential when implementing this kind of complex control systems [MLi07].

5 Software Quality Metrics

The intuition of the researchers is not sufficient to state about the quality of produced software or the efficiency and adequacy of the development approach. *Quality metrics* are necessary in order to demonstrate that lightweight formal methodology is successful in the digital hydraulics area; they are the evidence for scientific progress. Some metrics for control systems exist, but are limited to performance and simulation results. Simulink itself offers simple and direct model measurements.

By *quality* we mean conformance to requirements and fitness for use, both of which are related and consistent [Kan03]. We focus more on the former, as we approach the software creation process from the developers' point of view. Any nonconformance is considered as defect, thereby decreasing the quality of system.

To assess the developed system we are examining three of its aspects: *project* characteristics and execution, the final *product* and *processes* governing its development. Each of these is in fact a composition of many other, more detailed characteristics. The application of presented metrics is discussed in Section 6.

5.1. Project and Process Metrics

When analysing *project* metrics we take under consideration the resources used in the development, such as number of developers in the team and their skill levels, the structure of staff assigned to different tasks, the schedule and division of project lifecycle, as well as the effort. When analysing the resources, one has to also consider tools and methodologies used as they are one of the factors heavily influencing the quality as a whole.

In the examination of the development *process* one has to study the development itself and the impact, which this has on overall product quality improvement. The observations that are made are essential for the enhancement of software development and its later maintenance. Process metrics include the effectiveness of defects removal and time needed to fix discovered problem. By *defect* we mean an anomaly in a product and this term is used interchangeably with *fault* [Kan03].

When the numbers of detected and removed defects are known, we can compute the defect removal effectiveness (DRE), which is a process measurement. DRE is defined as the percentage of defects removed during particular phase per defects latent in the product. The latter is estimated by sum of defects removed during the phase and defects found later. In our work we are calculating the *defect removal phase effectiveness*, which is the DRE for specific phase of the development. The higher the value of this metric's outcome, the more effective is the development process. At the same time the defect propagation to later phases is reduced. Since in our development we are aiming for high reliability, DRE is one of the metrics that well describe the impact of using lightweight formal methods and contract-based design on our system's development. It is inaccurate to assume for all the defects to be injected into the system only at the early stages of the development. Observing the state of defects during development is crucial for portraying the development process itself, as well as the quality of the product at

every development stage. Having defect origin (the phase of defect introduction) as well as defect discovery and removal per phase we are able to gather information about the distribution of defects. Subsequently, from obtained data we produce a matrix of defect origin and discovery per phase. This cross-classification enables us to straightforwardly calculate various defect removal effectiveness measures for the specification, design and code inspection, along with unit and system test DRE. Moreover, the overall defect removal effectiveness for the entire development cycle can be computed, in order to examine defect detection efforts before the product is released to the field.

5.2. Product Metrics

The *product* assessment involves direct measurements about those physical features of the system, like size and structure measurements, that influence the complexity measurements. Product metrics can be used to describe performance and quality level. The size of the developed system will be measured in two ways, with respect to the Simulink environment.

The structures that are examined first, blocks, are gathered directly by the Simulink environmental command *'sldiagnostics'*. Block diagrams are the representations of the system in Simulink. Models created from blocks represent both data and the control flow, and can be simulated using Simulink. Since the implementation is diagrammatic, it gives a graphical view of the system and the development. It might also be used for the documentation, which in turn enhances systems maintenance.

The second ancillary size measurement is the Generated Lines of Code (*GLOC*) metric, understood as the number of physical lines, including executable lines, data definitions and comments, as well as blank lines and program headers. GLOC is not a fully reliable metric, since the automation does not allow measuring the productivity of programmer. Furthermore, it is negatively correlated with design efficiency, in which we are interested in. In our research GLOC serves to determine the relation between the diagrammatic block structures and the executable code itself.

GLOC and block attributes are used for discussion in the context of *defect* rate calculation in particular development phases. Defect density is relative to the software size and directly influences the system quality. We classify the defects by the phase of their origin and the phase that they were found. This enables us to analyse the distribution of defects over entire development process.

The *complexity* metric for the Simulink models is one of the main outcomes of our research. It is based on the structural and relational attributes of the product and influences not only the schedule of the project, but also the productivity of the developers. The higher the complexity, the more effort the developers need to make in order to design a system. In our research we use the *Card and Glass complexity model* [Kan03], which we have adjusted to Simulink environment [PlW07]. This metric evaluates a system by analysing its structure and properties of its model. Our complexity model consists of two components, i.e. *structural* and *data complexity*. For calculating both of the elements the structure called *fan-out* is required. Since the

Simulink system has a graphic representation and its blocks can be interpreted as modules, fan-out stands for a count of subsystems that are called by a given subsystem. In our case it is the count of subsystem blocks that are connected with given subsystem block by input-output parameter. In other words, it is the number of subsystem blocks that can be reached by leaving a given one. Structural complexity is defined as a mean of squared values of fan-out per number of subsystem blocks. Data complexity is specified as a function dependent on the number of inport/outport variables and inversely dependent on the number of fan-out in the subsystem block. After calculating complexity and analysing the results of the computations, we shortly present statistics about *library usage*. This is another measurement related to physical features of the system. It can be obtained with '*sldiagnostics*' command.

6 Quality Measurements in TC II

Both project and process characteristics influence the quality of final product. Resources and methodology used, as well as the development itself affect the software features, which can be assessed with a certain measurement program. Presented framework can be used for determining the quality of Simulink systems also in domains other than the digital hydraulics.

6.1 Project and Process Quality Measurements in TC II

The project in focus was created by three system developers and three indirectly involved project staff members. The development team is well-experienced in the (digital) hydraulics field and averagely experienced in formal methodologies (almost two years of practice). The latter is caused by cooperation with the formal researchers on preceding project. The Simulink is used as the development environment, in which the team members have already been programming for approximately two and a half, five and fifteen years correspondingly.

The iterative development of TC II is combining the superposition refinement with the contract-based design approach in order to obtain reliable and efficient digital hydraulic system. Each of the iterations relies on detailing previous development step, and at the same time it acknowledges the rules of used methodologies. The development is carefully planned; from the comprehensively created system specification, through the deliberately prepared design and controlled coding, to the testing. This lightweight formal approach influences the architecture of the system by structuring it into layers. Each successive layer has more detailed and more extensively specified behaviour than the preceding layer.

The development of the project was divided into five phases: specification, refinement, programming, unit and system testing. The first two can be interpreted as the design or modelling phase. Development with methodologies such as the contract-based design and stepwise refinement devote attention to creation of the system in such a way that it fulfils the requirements and preserves given properties. This in turn directs

the focus to the modelling phase and at the same time reduces the time needed for testing. The unit testing is done in an iterative style, which follows the idea of stepwise development. Therefore, each component is tested after being coded. The system testing phase starts once all components are coded and individually tested.

In our research we analysed the effort made in each development phase in order to be able to examine the development process. The development took 94 man-hours in total. The specification and refinement phase used majority of the development time (65%), whereas the programming phase took 13%. The remaining part was spent on testing. The time needed for testing is relatively small in comparison with projects that are not using formal approaches [JAH90] [LFB96] [JPB06]. The overall system testing phase (3%) is significantly shorter than the unit testing phase (19%), which indicates that final assembly of the system from the already tested components is effective and cost-efficient. The development phases are slightly overlapping with neighbouring ones, as they influence each other.

The documentation of the system and justifications of the design decisions were created simultaneously with the development of the system in its initial two phases. This increased the effort in the first phase by about 25%. However, it eliminated the need of writing the documentation after the deployment of the project, which is proven to be laborious and deficient [PlW07]. The approach to the documenting process we used is beneficial, as the information being recorded is complete, consistent and thoroughly checked. Moreover, we obtain better management over project by relying on good quality documentation. Additionally, the maintenance of the system is facilitated.

6.2 Product Quality Measurements in TC II

The produced system is the main outcome of the project and is analysed as such. First, we focus on the physical measurable feature of the system, which is the system's size. The Simulink environmental command '*sldiagnostics*' provides us with the number of blocks. We have 854 blocks in total. TC II contains 54% more blocks than the previous development, which was less complex and used lightweight formal methodology to a very little extent. The increase in number of blocks is caused by the fact that more functionality needs to be accomplished by the system. The code that was automatically generated from the model consists of 14 428 lines of code (GLOC). From these size measurements we are particularly interested in blocks, as we can examine their relation with GLOCs. The ratio between the number of GLOC and blocks is 17, which is comparable with previously developed systems [PlW07].

In the assessment of software product measurements, gathering information about defects provides a broader view not only on the quality of the system, but also the process of quality assurance. It serves for depicting the process of handling defects – discovering and removing them. In our research we are interested in the defect density and the distribution of defects over the development phases, focusing mostly on classification of defects with respect to their origin and detection phases.

One of our main goals of using contract-based design methodology in combination with stepwise refinement was to obtain reliable and correct software. Therefore, we are concerned with the calculation of defect density. We define defect density as a measure of the total known defects divided by the size of the software entity being measured. There were eight defects found throughout the development cycle of the system until its deployment. Considering relatively small system's size (854 blocks, which corresponds to 14 428 GLOC), the defect density for the final product is very low (0.0095 defects per block or 0.554 defects per kGLOC). For comparison, in previous development there were 0.0221 defects per block or 1.38 defects per kGLOC. It is worth mentioning that in earlier development the team was already using refinement approach and gaining knowledge of how to apply the contract-based design methodology to their development process. Low defect density in TC II was influenced by the skilled development team, well-experienced in digital hydraulics field, as well as generation of code from the model. The equivalent data from projects before the application of lightweight formal methods were not gathered. Therefore we are not able to perform a baseline comparison. Nevertheless, presented numbers give a solid evidence of the accuracy and adequacy of the used approach in a perspective of system's quality.

In Figure 3 we present the classification of defects throughout the system development. The origin of defects is placed in the specification and programming phases, whereas their detection – in programming, unit and system testing. It is worth mentioning that there was very small amount of defects found and the majority of them were discovered by programming and unit testing. Only one defect was found in the last phase of the development, when the system was assembled and tested as a whole. There were no defects found after the deployment of TC II; however, it has not been extensively used yet. For the same reason the tear and wear issues cannot be raised here.

During the system's development we mostly concentrated on the design phase in order to decrease the likelihood of introducing the defects in the early stage of the project, which in consequence would cost more to handle later on. By obtaining the desired behaviour of the system (contract-based design methodology) we prevent system's unplanned adjustments and augmentation. It would require significant amount of resources and might possibly lead to injecting new defects to the system. Since the produced system is correct by construction, the defect density is reduced and the possibility of defect propagation is minor.

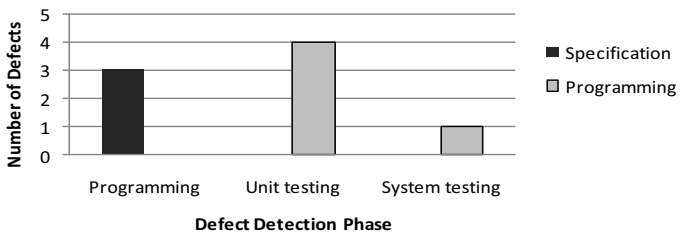


Fig. 3: Defect classification

Another defect-related metric is defect removal effectiveness (DRE), which is a process measurement. The higher value of this metric translates to more effective development process. Moreover, this means that fewer defects propagate or emerge in the next phase. Given that the defects were found and fixed in the programming, unit and system testing phases, the effectiveness of defect removal considers exactly these phases. In our calculations we did not take into account estimations concerning the after-deployment phase, as the system was not used extensively enough. The results are presented in Figure 4. In programming phase there were three defects removed, out of eight known, therefore the effectiveness is 37.5%. At the same time five other defects were injected during programming. The defects from the programming phase were removed in sequence: four during unit testing and one – during system testing phase. The defect removal phase effectiveness for those stages of system development is equal to 80% and 100% respectively.

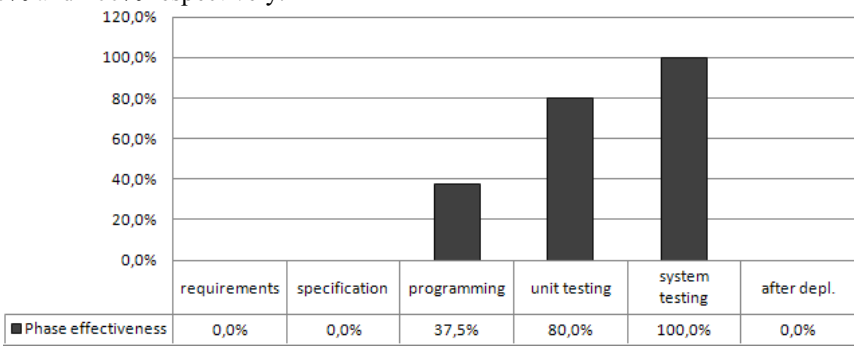


Fig. 4: Defect Removal Phase Effectiveness

When considering the physical characteristics of the system we also need to focus on the structural measurements, since the Simulink models are represented in a graphical manner. The model of the system is structured in a *layered* style. This means that when "entering" one subsystem block, more detailed lower layer is exposed. Our system is organised in five layers, where the uppermost one represents the controller and bottom one implements low-level computations. We focus on the structure starting from the second top-most layer, the Cost Function and Optimal Control, and afterwards its subsystems. We are interested in the structure of the system, as it gives information needed to assess the complexity of each layer.

In order to be able to compute the *complexity* based on our complexity model [PlW07], we need to obtain the number of inport and outport variables in the subsystems, as well as compute the fan-out values and count the number of subsystem blocks in examined layer. It should be noted that inport and outport variables used in the complexity calculations are represented in the Simulink diagram by in-going and out-going arrows to and from the subsystems of a considered layer, respectively. By Simulink definition, Inport blocks are the links from outside of a system into it and correspond to inputs, whereas Outport blocks are the links in the opposite direction

representing outputs. Therefore one Inport or Outport block (represented in the diagram as an oval with an arrow) can represent several inport or outport variables of the same value with respect to several subsystems that are using it. In Table 1 we have gathered all the structural measurements for the second and third highest layers.

Tab. 1: Structural measurements of the system – second and third layer

Layer level	Subsystem layer	Subsystem blocks	Fan-out	Inport blocks	Outport blocks
2nd	Cost Function and Optimal Control	5	8	17	4
3rd	Finding Optimal *u*	2	1	3	4
3rd	Pressure Terms	4	3	8	8
3rd	Secondary DFCU Terms	5	4	8	7
3rd	Switching Terms	5	7	3	9
3rd	Velocity Terms	3	3	3	4

Cost Function and Optimal Control subsystem is the subsystem in second highest layer and consists of five lower level subsystems: Finding Optimal *u*, Pressure Terms, Secondary DFCU Terms, Switching Terms and Velocity Terms. Because of that structuring we observe the relatively high number of fan-outs, as the functionality is deferred to subsystems at lower layers. Very high value of Inport blocks in the upper layer is caused by the tasks that they are performing and additional configurable user parameters, as well as parameters for tuning. The Outport blocks represent the outcome of the algorithm in addition to the fault tolerance mechanisms.

We follow the model formula for the structural complexity, $S = \frac{\sum_{i=0}^{n} f^2(i)}{n}$, where *f(i)* is fan-out of subsystem block *i* and *n* is a number of subsystem blocks in the system. Furthermore, we compute data complexity according to the formula $D = \frac{V(i)}{n(f(i)+1)}$, where *V(i)* is the number of inport/outport variables in a subsystem block *i*, *f(i)* and *n* are as mentioned before. The structural measurements necessary for computing complexity are gathered in Table 2. It represents the Cost Function and Optimal Control layer and its subsystems with corresponding values of fan-out, inport and outport variables, as well as the sum of inport and outport variables. Values for lower layers are gathered in the same manner.

Tab. 2: Structural measurements of Cost Function and Optimal Control layer

Cost Function and Optimal Control	Fan-out	Inport variables	Outport variables	Overall variables
Velocity terms	1	3	4	7
Pressure Terms	1	10	9	19
Switching Terms	1	3	9	12
Secondary DFCU Terms	1	8	7	15
Finding the Optimal *u*	4	11	4	15

Table 3 presents values of structural and data complexity, along with total complexity for the Cost Function and Optimal Control layer, as well as its subsystems. From these results we deduce that the Cost Function and Optimal Control layer has the highest value of structural complexity. This is caused by the fact that it is the higher level layer, which represents the abstracted view of all the lower layers. Furthermore, there are more connections between the subsystems (high fan-out), which also increases the structural complexity. Moreover, the Cost Function and Optimal Control layer is structured in a recursive manner. This means that the subsystem block where the optimal u is being found is called by other subsystem blocks in the layer. This forms a structural loop and in consequence increases the value of fan-out, which in turn increases the value of computed structural complexity.

Tab. 3: Complexity measurements

Subsystem layer	Structural complexity (S)	Data Complexity (D)	Total Complexity (C = S + D)
Cost Function and Optimal Control	4	5.9	9.9
Finding Optimal *u*	0.5	7.5	8
Pressure Terms	1.25	4.96	6.21
Secondary DFCU Terms	2.75	3.5	6.25
Switching Terms	5	2.62	7.62
Velocity Terms	1.67	2.11	3.78

The recursive feature mentioned previously has the influence also on data complexity, but in an inverse manner, since the fan-out value is placed in the equation as denominator. Another factor increasing the data complexity is the number of inport/outport variables. The number of user configurable parameters is higher (comparing to previous developments), as the system can be used for both 4 and 5 Digital Flow Control Units systems. There are also more parameters used for fine-tuning the system's performance. This overall augmentation in inport/outport variables is an evidence of more functionality that needs to be accomplished by the system, and, as a consequence, increase of data complexity. There are many computational requirements, as a result of relatively complex control algorithms.

It is worth noticing that subsystem Finding Optimal u has high data complexity, which is caused by extensive algorithmic computations. Conversely, the structural complexity is relatively low, since the computational functionality of this subsystem is not deferred into lower subsystems.

The total complexity is the sum of structural and data ones. It is highest for Cost Function and Optimal Control layer due to high value of structural complexity. The values of total complexities for the lower layers are comparable, which is caused by equal decomposition of the functionalities for particular layers. The total measured complexity increase is relatively small (68%), considering the system's enhancements,

such as user and tuning parameters, more complex control algorithm and broader functionality. The developers' perception is consistent with this result.

Our system uses a set of *library blocks*, information about which can be obtained by triggering the '*sldiagnostics*' command. These are the *digital hydraulics*, *dspstat3* and *simulink* libraries. *Digital hydraulics* library is an in-house library and includes components that are commonly used in controllers of digital hydraulic systems. It includes functions that are used to calculate the force equation of a cylinder. They are very often used in the control algorithms therefore they are considered as quite reliable and complex. The *dspstat3* library is responsible for the display of mathematical and statistical functions such as correlation, standard deviation, minimum or maximum and sorting, among many others. The *simulink* library is built-in, generic type of library that can be extended according users' needs. In our system we are using seven links to the basic functions of the *digital hydraulics* library, one link to the *dspstat3* library to compute minimum and six links of the *simulink* library responsible of logic and bit operations.

An outcome of the development, the documentation, increases the quality level of maintenance process. Every design decision is documented and thereby justified. In addition, structuring the system into layers makes the system more modular and maintainable. Thus, the system is more modifiable, anticipating prospective changes, no matter if they are just adjustments or augmentation.

7 Conclusions

In our research we analysed the final software and its development process in order to be able to state about their quality. To our knowledge there are no metrics that could be used directly for control systems software. The existing ones concern system's physical performance and its simulation. The available in-house Simulink metrics concern mostly the straightforward statistics about the model, as well as design effort and time related metrics. Since software is becoming more composite nowadays, there was a need of more elaborate metrics, such as complexity or defect related metrics. We presented these metrics in a perspective of lightweight formal method development in the Simulink environment. We illustrated how the metrics can be applied to a digital hydraulics domain project, TC II. Our framework, however, is more flexible and we believe it can be used in other domains as well.

In our future work we would like to extend the proposed complexity metric to those low-level block structures that are not subsystems. A normalisation of the planned complexity metric with other model factors would give a comprehensive view on this characteristic. Additionally, we aim at continuing the research on the impact of lightweight formal methods on the quality of the development process and product. Since security and safety related issues were not in the scope of this paper, we consider them as one of the possible directions of our future exploration.

References

[BaS96] Back R.J.R., Sere K., *From modular systems to action systems*, Software - Concepts and Tools 17. (1996)

[BaW98] Back R.J.R., von Wright J., *Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science*. Springer, Heidelberg (1998).

[BKS83] Back R.J.R., Kurki-Suonio R., *Decentralization of process nets with centralized control*, 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. (1983)

[Bos07] Boström P., Linjama M., Morel L., Siivonen L., Waldén M., *Design and Validation of Digital Controllers for Hydraulics Systems.*, 10th Scandinavian International Conference on Fluid Power (SICFP'07). (2007)

[Bos08] Boström P., *Formal Design and Verification of Systems Using Domain-Specific Languages, PhD Thesis.*. TUCS, Turku (2008).

[Fen00] Fenton N., Neil M., *Software metrics: roadmap.*, Conference on the Future of Software Engineering. , Limerick, Ireland (2000)

[Fen97] Fenton N., Pflegger S., *Software Metrics. A Rigorous and Practical Approach.*. PWS Publishing Company (1997).

[Gra94] Grady R., *Successfully applying software metrics*, Computer, 27 (1994), pp.18-25.

[JAH90] Hall J.A., *Seven Myths of Formal Methods*, IEEE Software, (1990), pp.11-19.

[Jor99] Jorgensen M., *Software Quality Measurement*, Advances in Engineering Software, 30 (1999), pp.907-912.

[JPB06] Bowen P., Hinchey G., *Ten Commandments of Formal Methods Ten Years Later*, IEEE Software, (2006).

[Kan03] Kan S., *Metrics and Models in Software Quality Engineering*. Addison-Wesley (2003).

[Kat93] Katz S.M., *A superimposition control construct for distributed systems*, ACM Transactions on Programming Languages and Systems, (1993), pp.337-356.

[KRK05] Kandt R., *Software Engineering Quality Practices*. Auerbach Publications (2005).

[LFB96] Larsen P., Fitzgerald J., Brookes T., *Lessons Learned from Applying Formal Specifications in Industry*, IEEE Software, (1996).

[Lin05] Linjama M., Vilenius M., *Digital Hydraulic Tracking Control of Mobile Machine Joint Actuator Mockup*, SICFP'05. , Linköping, Sweden (June 2005)

[Lin08] Linjama M., Huova M., Vilenius M., *Online Minimisation of Power Losses in Distributed Digital Hydraulic Valve System.*, The 6th International Fluid Power Conference. , Dresden (2008)

[LiV07] Linjama M., Vilenius M., *Digital hydraulics - towards perfect valve technology.*, The 10th Scandinavian International Conference on Fluid Power. (2007)

[LKV03] Linjama M., Koskinen K.T., Vilenius M., *Accurate tracking control of water hydraulic cylinder with non-ideal on/off valves*, International Journal of Fluid Power, 4 (2003), pp.7-16.

[LMB09] Lindemann U,, Maurer M., Braun T., *Structural Complexity Management. An approach for the Field of Product Design*. Spinger (2009).

[Mat07] (MAAB) MathWorks, *Control Algorithm Modeling Guidelines Using Matlab®, Simulink®, and Stateflow®, Version 2.0*. (2007).

[Mey92] Meyer B., *Applying Design by Contract*, In Computer (IEEE), vol 25, no. 10, (1992), pp.40-51.

[MLi07] Linjama M., Huova M., Boström P., Laamanen A., Siivonen L., Morel L., Waldén M., Vilenius M., *Design and Implementation of Energy Saving Digital Hydraulic Control System*, SICFP'07. , Tampere (2007)

[PlW07] Pląska M., Waldén M., *Quality Comparison and Evaluation of Digital Hydraulic Control Systems*. Turku Center for Computer Science (TUCS), Turku (2007)

[Sii05] Siivonen L., Linjama M., Vilenius M., *Analysis of Fault Tolerance of Digital Hydraulic Valve System*, Power Transmission and Motion Control. (2005)

[Sim09] Simulink - Simulation and Model-Based Design, http://www.mathworks.com/products/simulink/

Publication 5

Simulink-Specific Design Quality Metrics

Marta Olszewska (Pląska)

Originally published as: TUCS Technical Report 1002, Turku (Finland), March 2011.

©2011 Turku Centre for Computer Science (TUCS). Reprinted with kind permission of TUCS.

Simulink-Specific Design Quality Metrics

Marta Olszewska (Pląska)
Åbo Akademi University, Department of Computer Science
Joukahaisenkatu 3–5, 20520 Turku, Finland
marta.plaska@abo.fi

TUCS Technical Report
No 1002, March 2011

Abstract

High quality of a software system is important at every stage of its lifecycle. For evaluation purposes some unambiguous measurement mechanisms, which are appropriate for a certain setting and development phase, are needed. We focus on the assessment of the design of a system that is modelled in the Simulink environment. This paper introduces several metrics that support the model construction in Simulink by controlling interdependencies of its hierarchical structure and its complexity. The measurements are providing the developers and system architects with the data that can facilitate their design decisions and indicate possible design problems. We regard our measurements as a method to improve the quality of the design and, by that, the quality of the overall system.

Presented metrics are applied to a case study from the digital hydraulics domain. They are investigated in order to empirically verify their suitability to support the modelling activity.

Keywords: Quality assessment, measurements, complexity, interdependencies, Simulink, model construction

TUCS Laboratory
Distributed Systems Design

1. Introduction

Use of measurements and metrics for comparison, improvement or simply for evaluation purposes helps to build scientific knowledge in many areas. There is a common need to be able to relate to particular characteristics in a quantitative and qualitative manner in everyday life. Also in engineering disciplines and software development we observe a large number of measurement models that are adequate for a specific application or domain.

In the software field there are many well known product metrics, just to mention the ones destined for software coded in OO languages, or process metrics, which are being adapted to certain development setting. Nevertheless, there is a need for metrics for the early stage development measurements [1] Moreover, there is a gap in terms of metrics dedicated for domain specific tools. Therefore, there is a tendency to shift the focus from end-phases of the development to the construction of the software system at its design phase, which also takes into consideration the specificity of the development environment.

In this paper we propose complexity and interdependencies measures and identify them as the indicators of design quality. These measures enable the dynamic tracking of the high-level design issues at the modelling stage and indicate possible improvements. Hence, they provide an early control over the quality of the project. Monitoring the development at the modelling phase gives an insight on the prospective quality of the design and final software [2]. Hence, managers obtain certain degree of control over the project and anticipate the allocation of resources. Furthermore, developers are provided with the valuable information indicating e.g. too large or too complex parts of the system, which in sequence can imply the potential design problems and unnecessary complication in later development stages. Management of the identified issues and performing modifications in the initial phase leads to cost and effort savings [3] [4]. Model inspection and redesigning activities that are indicated by measurements, as well as different decomposition of the functionality of the system, benefit in better quality of the final system.

The rest of the paper is structured as follows. In Section 2 we motivate the need of performing measurements for Simulink modelling environment and give rationale for evaluating Simulink-specific models from the software perspective. Section 3 gradually introduces the metrics, first the complexity related ones, then those based on model interdependencies. In each case we depict the meaning of the metric, as well as its objective. Some experimental results based on a case study are presented in Section 4, where we also give evaluation of our metrics with respect to the test case. We describe related work in Section 5, whereas in Section 6 we conclude and present directions for our future work.

2. Measurements for Simulink

Our conjecture is that the control over the development given by measurements is particularly important when it comes to modelling large, dynamic and embedded systems, like control systems, since they need to be both reliable and correct. In our research we use MATLAB/Simulink [5] – a high-level graphical modelling environment for the purpose of model-based design and multi-domain simulation. This powerful tool is appreciated in industry, especially in scientific and engineering areas, as it offers an interactive visual setting and can be configured according to the domain-specific needs. It allows the design of the system, simulation of the created design, code generation and various testing options in an intuitive and easy to comprehend manner.

We focus on supporting the modelling activity in Simulink with measurements that are specific to this environment. Our motivation is to enable evaluation of the system and its characteristics from software perspective at the high-level design stage of the development. The justification for our investigation is that there is a need for software-oriented measures for the Simulink environment, since the current quality evaluations mostly concern the results of model simulation or system's performance assessments. System engineers, who base their success on years of experience, can facilitate their development methodology with measurement-supported approach.

2.1. Software Perspective in Simulink Setting

We are especially interested in measurement-supported creation of systems that have a hierarchical structure, where each lower layer represents the system in a more detailed way. Systems with this structure, due to their nature, tend to be reusable and easy to extend and modify [6] [7]. Such systems are good subjects for application of measurements as means for quality improvement. Simulink enables a layered type of modelling by e.g. masking of certain structures, decomposition of the functionality and encapsulation of specified subsystem elements.

A Simulink model is represented by dataflow diagrams, which consist of (functional) blocks connected by signals via the connection points called ports. Since the blocks have memory, their output values depend on the values of the preceding input. The (functional) blocks can be organised into subsystem blocks in order to facilitate modelling, as well as enable hierarchical design. Such subsystem blocks can have any number of ports for input and output of data to and from the system, respectively. For the flexibility purposes, blocks can be parameterised.

In Figure 1 we show an example of a Simulink diagram [8], which computes the length of the hypotenuse in a triangle using Pythagorean theorem. On the left hand side we observe the subsystem block *Pythagoras* with two inports and two outports. On the right hand side we present the diagram that consists of two subsystem blocks: *Calculate* and *Check*. It is in fact a lower layer for the subsystem *Pythagoras*. The rectangular elements with smoothed corners match to the inports and outports of the subsystem that contains them, whereas the lines with arrows represent the signals.

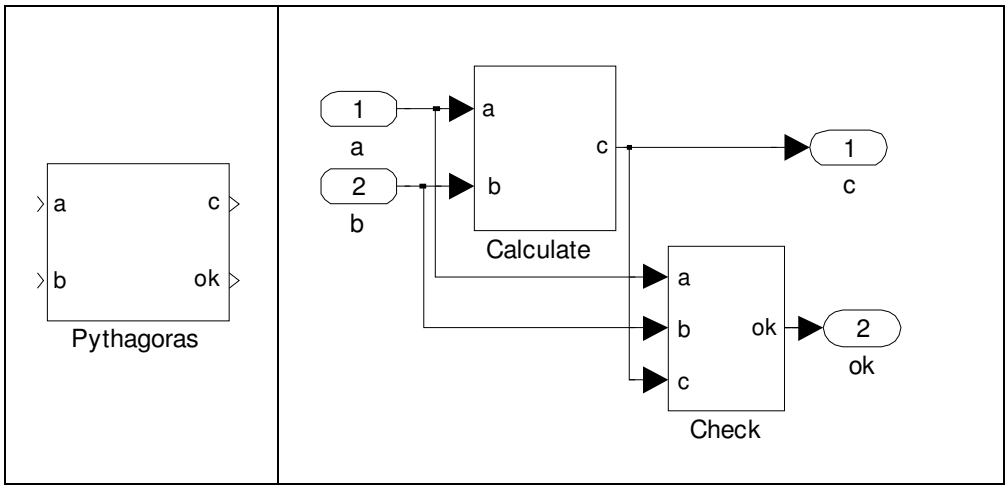


Figure 1. Example of Simulink diagram

In our previous work a problem of measuring the impact of rigorous contract-based design development methodology [9] on the quality of Simulink model and the final system was investigated. An analysis was performed to evaluate the maintainability attribute from the perspective of software construction. The focus was on the complexity characteristic, since it is a vital factor when it comes to reliability [10], understandability, extendibility, reuse, cost and effort of development and maintenance of the system [11].

In our previous research a Simulink-specific complexity model was created, which considered not only the structural, but also data complexity of the constructed system [12]. The model was validated against several case studies from the domain of digital hydraulics [13] and is compliant with the perception of the developers.

The complexity model is a starting point for the research presented in this paper; therefore, it is necessary to shortly describe its underlying principles. The model is based on several fundamental Simulink diagram elements, as well as the relations between them. For the computation of the *structural complexity S* of a layer *l* with *n* blocks the number of fan-outs *f(b)* of each block *b* is needed. Fan-out is a count of subsystem blocks that are called by a given subsystem block *b* within the same layer. Structural complexity is described by the following formula:

$$S(b) = \frac{\sum_{b \epsilon l} f^2(b)}{n} .(1)$$

Data complexity D, on the other hand, is additionally dependent on the number of inport and outport variables *V(b)* of the subsystem block *b* in the examined layer and is given by the formula:

$$D(b) = \frac{V(b)}{n \cdot ((f(b)+1)}. (2)$$

The *complete complexity* model *C* is specified as a sum of structural *S* and data *D* complexities and is used as such in the rest of this paper.

Our current research derives from the established complexity model and extends the measurement scope presented in [14] and [12]. We further analyse the impact of the complexity attribute on the Simulink model by examining it on a layer-by-layer basis. The basic constructs used are layers, (functional) blocks, subsystem blocks, incoming and outgoing signals (contributing to the number of fan-in, fan-out, in/output variables),

which represent the logical view of the Simulink design. An Eclipse-based tool to support contract-based verification of Simulink models has been created and is being further extended [15]. In order to enable automatic data collection and computing the measurements during the modelling of the system a plug-in to this tool has been developed. This allows us to give continuous feedback about the model of the system during its high-level design.
We have constructed several derived metrics, which help us to shape the modelling strategy for the Simulink environment developments. We regard our measures as preventative, since they indicate the design issues that are and can be problematic in later stages of system's lifecycle. By observing the complexity related measures and measures associated to hierarchical design we analyse whether the subsystem needs some design changes. This way, by applying horizontal or vertical layering (abstraction and decomposition, respectively) we bring reinforcements to the battle with growing complexity of (large) systems.

3. Quality Measures

The metrics we create in our current work are meant to indicate the potential defects or weaknesses of the high-level design. The objective is to minimise the impact of poor design issues on quality of the system by handling the problems directly at the modelling stage. This involves an analysis of the system with its design measurements, since at this point of the development it can be done most cost-effectively.

3.1. Complexity Controlled

The first metric that we established for Simulink is the *weighted block count per layer* (*WBCL*). It resembles one of the object-oriented metrics given by Chidamber and Kemerer in [16], where the sum of the static complexity of all methods in a class was in the scope. WBCL measure represents a weighted sum of blocks in a given layer. Possible weights could be size (number of subsystem blocks or generated lines of code), McCabe cyclomatic complexity [17] or simply weight=1, designating that this is a non-weighted block count per layer. When considering the latter, WBCL can be treated as a measure of size. In our research we chose the complexity model created in our previous work [12] as a weighting factor. Hence *WBCL* for a layer *l* is represented by the formula

$$WBCL = \sum_{b \epsilon l} C(b), \quad (3)$$

where *b* is a certain subsystem block in a layer *l* and *C* is a complexity of given block *b* computed according to our complexity model presented in Section 2.1. The results of the *WBCL* computations enable us to compare the complexities of each layer in the Simulink model, as well as allow further calculations. The higher the value of the measure is, the more complex the layer.

WBCL characterises the complexity of a subsystem blocks in a layer as a whole. It can be used as a higher-level indicator of the development and maintenance effort for the cross-layer comparison. Subsystem blocks with a large WBCL value can be redesigned

(decomposed) into two or more subsystems. The decision about design changes should be made by the experienced developer or a system architect, who is supposed to give his reasoning in the appropriate design documents.

We derive *average block complexity per layer* (*ABCL*) measure to be able to identify the outliers that indicate too low or too high complexity. *ABCL* is described by the following formula

$$ABCL = \frac{WBCL}{n}, (4)$$

where *n* is the number of blocks in a layer.

This metric allows the design team to assess the Simulink diagram more closely, from the perspective of a particular layer. Having the average block complexity per layer, the distribution of complexity in a layer is more apparent. The developers can already at this stage focus the design inspections, concentrate their review and testing resources on those subsystem blocks that (in their experience) have the greatest possibility for improvement.

In order for the metrics per se to be beneficial for the development team and the managerial staff, we use ABCL in further computations. As the next step, we compute sample standard deviation of a layer according to the formula

$$s = \sqrt[2]{\frac{1}{n-1}\sum_{b\in l}(C(b) - ABCL)^2}, (5)$$

where *s* is the standard deviation, *n* is the total number of subsystem blocks in a layer *l*; *C, b* and *ABCL* are as defined previously. Our goal is to examine how much variation there is from ABCL. This will enable us to examine the high-level design of the system from the point of subsystem block.

We adapt to our setting and apply the "Goldilocks Principle", which is used in e.g. economy or biomedicine. It generally states that some characteristic must fall within certain margins, as in contrast to reaching extremes. In our case the size and complexity of subsystem blocks are the attributes that should be "feasible", meaning that they should not be too small or simple and too big or intricate, when compared to *typical* artefact. We propose to use standard deviation as in (5) in order to determine what is *typical*.

Particularly, the situation where the subsystem block complexity seems disproportionally high, compared to the complexity of other subsystem blocks, is of interest. The main reason is that it is more probable to be prone to defects. Moreover, it will potentially become system-specific, which in consequence means less reusable. Complex subsystem blocks are not only harder for the developers to understand and maintain, but also harder to test. This in turn means less model coverage. Since the complex subsystem blocks often have more interactions, they tend to cause more problems.

We compute the interval (*ABCL-s*; *ABCL+s*), which includes the subsystem blocks of desirable complexity values. This way we are able to observe the *outliers* that are placed outside the interval. On a distribution plot these are the left and right hand side ends,

which in our case represent the fragile subsystem blocks. These blocks should be carefully examined and possibly redesigned.

We use fat-tail distribution to point out the subsystem blocks that are "outside the margins". This distribution has been proven to be present nearly everywhere in software measurements, in particular in the Chidamber and Kemerer metrics [16] [18] [19] [20]. The "U curve" distribution of defects in a system has already been described by Hatton [21], where the author showed the relationship between defects and size of a component. Here we concentrate on the bell curve type of relationship of the distribution of defects and the complexity of (subsystem) blocks.

Table 1. Complexity of blocks versus quality problems

complexity of blocks / defects	many	few
too high	coding + design	design
normal	possible coding errors	OK.
too low	coding + design	design

Our observations are shown in Table 1, where the high-level design is handled with respect to the outliers methodology we described in this section. Too high or too low complexity of blocks in combination with many defects signifies design and coding problems, whereas when having few defects it denotes mostly design issues. In case the complexity of a block falls into the margins set by our metric the problems can be purely caused by coding (many defects scenario) or we consider that no problems are indicated (only few defects found).

One should keep in mind that it might also be the case that the high-level design of a given subsystem block, although pointed out as tenuous by measurements, is supposed to be left unchanged. It can be caused by the judgement of the designer or architect that this part of the system will not be extended or reused in the future, the redesign is not worth an effort or there is no time or resources for the changes.

3.2. Interdependencies – Keeping the Balance

In our study we additionally examine and analyse measures related to model hierarchy (nesting) and relations between the system layers. Our motivation is that horizontal layering is useful when dealing with complexity via abstraction mechanisms. However, one should note that multiple nesting makes the design harder to comprehend. We investigate these design characteristics, as they impact the understanding of the design, size and the overall complexity of the system [22].

We were inspired by the metrics created for object-oriented languages. Robert C. Martin [23] established several package metrics evaluating the conformance of a design to a pattern of dependency and abstraction. Since we want to be able to assess the cross-layer relationships between the subsystem blocks, we introduce the concept of coupling measures to the Simulink model design. The measures are based on the dependencies between abstract and concrete units in the model, i.e. subsystem blocks and functional blocks. In our research we investigate two characteristics of the design: instability and abstractness.

For our setting we define the primitives and specify the meaning of the relations between them. We deal with Simulink block diagrams which consist of layers, where each layer is defined by subsystem blocks or functional blocks. The subsystem blocks are present for the purpose of facilitating the organisation and understanding of a diagram. Otherwise, they have no impact on the meaning of the diagram and do not define a separate one; they can be treated as an abstraction of several blocks.

Subsystem and functional blocks depend on each other within one layer through incoming and outgoing signals. We define *afferent coupling between blocks* (*CaB*) as measure of the total number of external blocks linked to a given block due to incoming signal within one layer. In other words, it is the number of destination blocks for the block under analysis. *Efferent coupling between blocks* (*CeB*) is defined as the number of blocks that are linked to a given block due to outgoing signal within one layer, i.e. it is the number of source blocks for the given block. In both measures each dependent block is counted only once, regardless of the number of signals linking it to a given block. The count is equal to zero if the block is not connected to other blocks by incoming or outgoing signals.

Higher number of couplings between blocks signifies that the blocks are more dependent, which means that they rely on the outcome of other blocks. These blocks are also responsible, i.e. they are heavily dependent upon. Both of these features lead to the so called ripple effect, which means that introducing any change to the block has a considerable effect on other blocks, forcing additional changes. Moreover, high number of interactions stands against modularity and reuse, as well as augments the maintenance costs. Additionally, high coupling points towards high complexity and thus increases the costs of testing. It also denotes that the high-level design requires meticulous analysis and inspections, as it is troublesome to understand the existing dependencies.

Instability I of a (subsystem) block is defined as the number of efferent couplings between blocks divided by the sum of afferent and efferent couplings between blocks, which is given by the equation:

$$I = \frac{CeB}{CeB+CaB}, \quad (6)$$

where *CeB* and *CaB* are defined as before. The values of this metric range from zero (no incoming signals) to one (only incoming signals), where $I = 0$ denotes a completely stable (subsystem) block, and $I = 1$ signifies a maximally instable (subsystem) block.

Since instability is a characteristic defined with coupling measures, it impacts similar qualities of the system, i.e. modifiability, reusability and understandability. It is a sign for the developers to do the inspection of a given part of the high-level design and possibly make some changes. The (subsystem) blocks that are stable (I=0) have two features: they are simultaneously responsible and do not depend on other blocks. Same rule has been applied in object-oriented programming, where interfaces are those entities that are fully abstract and, as such, should be stable and not altered.

Instability was discussed system-wise for fault prediction in [24], where software design was considered from the risk-based analysis and verification. In our work we do not

propose any prediction mechanisms, but we use instability for more accurate model assessment through further computations and analysis.

Instability characteristic for Simulink was also mentioned in perspective of failure mode and effects analysis (FMEA) of automotive embedded software systems in [25]. However, authors derived their results mostly from the instability metric of a complete system, which was based on different primitives than we use in the paper. Furthermore, the granularity of our research is higher, since we focus on the blocks in the particular layers of the system.

We define *abstractness A* of a block as a ratio of the number of contained abstract (i.e. subsystem) blocks (*NaB*) to the total number of blocks in a layer the given block represents (*NB*), which is given by formula:

$$A = \frac{NaB}{NB}. \quad (7)$$

The presented metric has the range <0..1>, where 0 denotes a concrete block and 1 represents a completely abstract block.

In case the block is a leaf block we define the abstractness to be zero, i.e. the block is concrete. According to the metric the entirely abstract blocks will have the abstractness equal to 1, while in case of blocks containing only the functional blocks, or leafs in the lowest level of Simulink model, *A* will equal 0.

According to MAAB style guidelines [26], every level of a model should be designed with blocks of the same type. This means that the diagram should be structured so that the subsystem blocks are not mixed in one layer with functional blocks. In principle, the abstractness rule from object-oriented programming is advised to be used in Simulink model design, placing the subsystem blocks in one layer and basic blocks in a separate one.

Abstractness as a characteristic impacts the understandability and readability of the model. To make a subsystem block more abstract means hiding some of its details that are unnecessary at this level of comprehension. It also signifies that such subsystem block can be handled in more uniform way, which impacts maintainability and reuse.

Completely abstract subsystem blocks (*A*=1) should be stable, however its dependencies should be at some point concrete and instable to assure flexibility and modifiability. Furthermore, these dependencies should not have any more dependencies.

As it already has been confirmed for object-oriented methodologies, there is a relationship between stability and abstractness. In this paper we have indicated such a correlation to be true in the Simulink environment, as well. In order to confirm our claim, we identify the *distance* metric *D*, which we represent graphically as a chart (Fig. 2), where we take instability *I* and abstractness *A* as horizontal and vertical axis respectively. *D* is computed as a normalised sum of these values decreased by 1, which is given by the formula:

$$D = |A + I - 1|. \quad (8)$$

The resulting values range from *D*=0 to *D*=1, where the desirable values oscillate around *D*=0.

The *D* metric depicts the correlation between abstractness and stability for a (subsystem) block; it is an indicator of balance between these two characteristics. Therefore, when the distance exactly equals zero, it signifies that the balance between these two is optimal. According to the formula the desired situation is when the subsystem blocks are either totally stable and abstract (scenario *I*=0 and *A*=1) or entirely instable and concrete (*I*=1 and *A*=0). In Fig. 2 it is illustrated as a solid line, called in object-oriented languages "the main sequence" [23]. All the blocks that have a value of *D* close to the main sequence (between dashed lines) are said to have a good design.

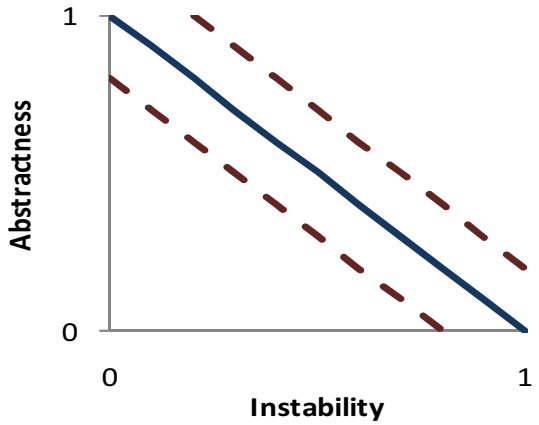


Figure 2. Graph representation of the distance metric D

Since when modelling in Simulink we want to follow the guidelines concerning the abstraction given by MAAB, the value *D*=0 will not be an optimal one. We have to set a margin fluctuating around the value proposed by Martin [23], to truly indicate that a (subsystem) block is coincident with the optimal balance with respect to its stability and abstractness. We called this a "zone of acceptance". Its value should be project-specific. On the other hand we can safely state that the value *D*=1 means that the high-level design is far from the desired characteristics and it should be carefully reviewed, and possibly restructured.

On the graph these problematic areas are the lower-left quadrant of the chart called the "zone of pain" and upper-right quadrant named "zone of uselessness". The former stands for the case when the (subsystem) block is very stable, i.e. has a lot of dependencies, but it's not extensible, i.e. has no abstract subsystem blocks. The latter represents a situation when a block is abstract, very extensible, but nothing depends on it, i.e. it is not used. This information highlights the priorities for the inspections and potential modifications of high-level design.

4. Experimental Results – Translating Findings into Insights

We applied the metrics presented in this paper to the case study from the digital hydraulics domain. The system under examination is named Test Case II and is a part of

a digital flow control valve. It models the cost function of the digital hydraulic valve controller via the subsystem Cost Function and Optimal Control (CFOC).

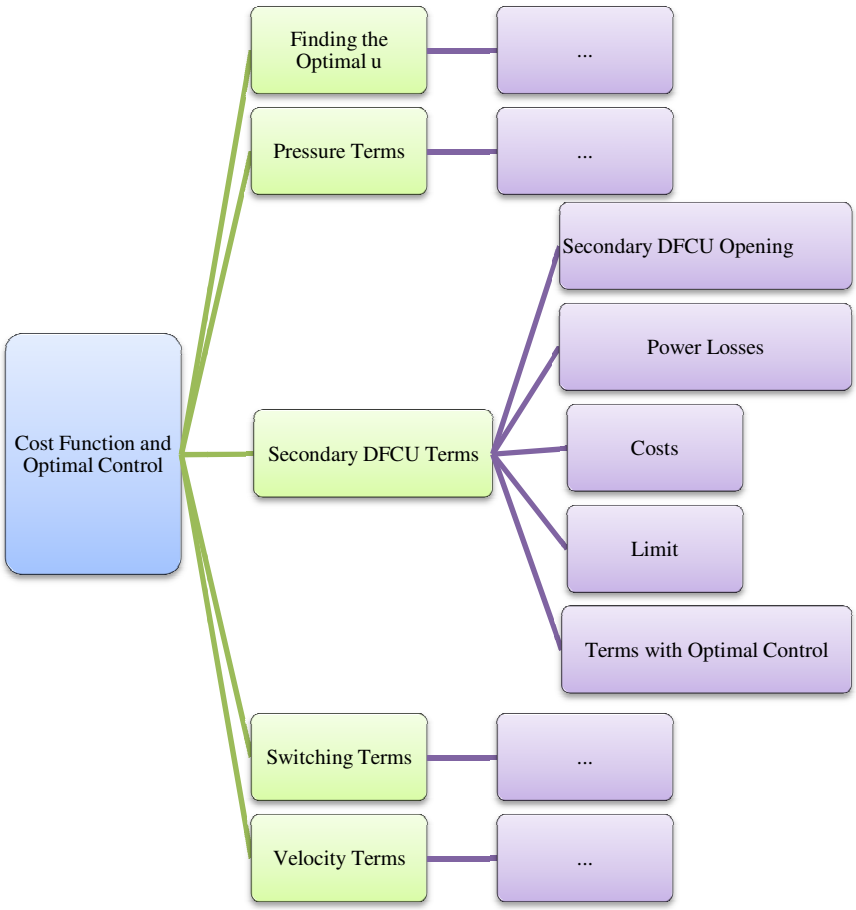


Figure 3. Structure of the Test Case II

The objective of the development was to choose the optimal control signals for the on/off-valves and fulfil some additional energy saving requirements. The model was developed using the lightweight formal Contract-Based Design method for Simulink [8] and consists of 846 blocks, which is represented by 14 428 automatically generated lines of code (GLOC).

The main subsystem CFOC consists of five dependent parts: Finding the Optimal u, Pressure Terms, Secondary Digital Control Flow Units (DFCU) Terms, Switching Terms and Velocity Terms, each of them consists of more subsystems. The hierarchical structure of the Test Case II is presented in Figure 3.

Due to copyright issues we are not permitted to present the Simulink diagrams of the model.

4.1. Case Study - Test Case II

We collected the data concerning complexity and interdependencies in the Test Case II. In Table 2 we present the complexities of the subsystem blocks specified by the subsystem CFOC, which are computed according to our complexity metric.

Table 2. Complexities of the subsystems of CFOC

Subsystem Name	Structural Cpx.	Data Cpx.	Total Cpx.
Finding the Optimal *u*	0,50	7,50	8,00
Pressure Terms	1,25	4,96	6,21
Secondary DFCU Terms	2,75	3,50	6,25
Switching Terms	5,00	2,62	7,62
Velocity Terms	1,67	2,11	3,78

Since we want to find indicators of bad design, we follow the steps presented in Section 3.1. First we compute weighted block count per layer (*WBCL* = 31.86) and then the average block count per layer metric (*ABCL* = 6.372). Next we calculate standard deviation (*s* = 1.66) in order to compute the margin (*ABCL-s*; *ABCL+s*), which for the CFOC is (4.72; 8.03). Finally, we identify the outliers, i.e. the subsystems that have too high or too low complexity. In our case the only part of the high-level design indicated as problematic is the subsystem Velocity Terms, since its complexity is too low. This subsystem is too simple and might be considered by the developers to be included in some other subsystem in the same layer. The line of reasoning for it to stay unchanged is the separate functionality or reuse purposes.

All of the subsystems in the CFOC layer are completely abstract. In order to give an interesting example on how the interdependencies metric works, we use one of the subsystems of the CFOC, i.e. the subsystem Secondary DFCU Terms. It consists of five subsystems: Secondary DFCU Opening, Power Losses, Costs, Limit and Terms with Optimal Control.

In Table 3 we present the number of afferent and efferent couplings between blocks, *CaB* and *CeB* respectively, which are obtained directly from the Simulink diagram. We also present the computed instability measure *I*. We observe that there are two subsystems that are totally stable (Secondary DFCU Opening and Power Losses), and two that are completely instable (Limit and Terms with Optimal Control). The former subsystem blocks are independent in a layer and responsible for the input to other blocks in the layer. The latter depend on the output of some other blocks in a layer, but they have no dependent blocks in the layer. The subsystem Costs is indicated as problematic within the high-level design of the subsystem Secondary DFCU Terms. It should be avoided as a subject for reuse, as it is responsible, but not independent.

Table 3 also portrays the abstractness metric *A* for the subsystem Secondary DFCU Terms. Number of abstract blocks *NaB* and blocks *NB* displayed in the table are the direct measurements acquired from the diagram.

Table 3. Measures for the subsystem Secondary DFCU Terms

Subsystem Name	**CaB**	**CeB**	**I**	**NaB**	**NB**	**A**
Secondary DFCU Opening	2	0	0	2	2	1
Power Losses	3	0	0	0	11	0
Costs	1	2	0,67	3	3	1
Limit	0	1	1	0	7	0
Terms with Optimal Control	0	3	1	2	5	0,4

It can be noticed that there are two entirely abstract subsystem blocks (Secondary DFCU Opening and Costs) and two utterly concrete blocks (Power Losses and Limit). This structuring is in accordance with the MAAB modelling guidelines. The subsystem

Terms with Optimal Control is the only one with a mixture of the block abstraction types, so it should be considered for inspection and possibly redesigned. The observations considering the meaning of abstractness measurement correspond to the general idea of the design in Simulink diagrams.

Table 4. Distance metric for the subsystem Secondary DFCU Terms

Subsystem Name	I	A	D
Secondary DFCU Opening	0	1	0
Power Losses	0	0	1
Costs	0,67	1	0,67
Limit	1	0	0
Terms with Optimal Control	1	0,4	0,4

Our aim is to indicate the high-level design problems through the use of the measurements. A complete picture of design issues can be seen by calculating the distance metric *D*, which is shown in Table 4. The value that defines the zone of acceptance is project-specific and was set to be 0.4. One should note that this threshold is experimental, since the general understanding of threshold identification and use in the metrics research area is not in the scope of this work. Therefore, every subsystem having the value D from 0 to 0.4 is considered as acceptable. From the results presented in the last column we can distinguish two subsystems with the optimal value of distance (Secondary DFCU Opening and Limit) and one subsystem that can be considered as fitting within the acceptance margin (Terms with Optimal Control). These may remain unchanged, since according to our metrics they fulfil the established design criteria. There are also two subsystems that can be indicated as having some design issues, i.e. Costs and Power Losses. The former is too instable and completely abstract, which contradicts the main idea of the good high-level design. The latter is neither abstract, nor stable to represent a high-quality design.

4.2. Threats to Validity

We consider four types of threats to validity [27]: conclusion and construct validity, as well as internal and external validity. The construct validity defined as ability to measure the object under study has been ensured by having a separation of concerns regarding the roles in the project. The person responsible for measurements was not involved in the Test Case II development; therefore, the objectivity of the measurements was preserved.

Since we have applied our methodology to only one (yet large) project, we have no statistical inference to approve our expectations. This is a threat to conclusion validity, as we are not able to draw any sound observations about out metrics based on the numerical deduction. Hence, the external validity is only partially achieved.

Test Case II example gives us the basis to generalise the results only to some degree. The internal validity defined as ability to separate and classify features influencing the examined variables without the researchers awareness has been reduced by discussions with Test Case II developers and Simulink expert.

5. RELATED WORK

There are several types of measurements on the Simulink tool level, just to mention basic data about the model. Some diagnostic information about a modelled system can be obtained with a command "sldignostics" [28], which allows collection of several direct model measurements, like the number of subsystem blocks, states, outputs, inputs, libraries, sample times of the root model and the memory used for each compilation phase of the root model. It also needs to be mentioned that the modelling standard checks are possible within Simulink. Therefore an intuition about the model is provided for the engineers; however, it is not sufficient when it comes to the analysis of the high-level design quality.

The content of Simulink (and also Stateflow [29]) model can be quantitatively measured with Modelling Metric Tool [30] described in [31]. This tool collects information about productivity and effort involved in developing a model and is customisable when setting the effort weights according to the difficulty of the architecture of the model. Moreover, it provides a graphical front-end for the information provided by "sldiagnostics" command and produces basic reports. Furthermore, several advanced model coverage measures are provided within one of the Simulink toolboxes. The abovementioned measures describe the development process and the model properties either in a relatively unsophisticated manner or from the testing perspective. The detailed evaluation of the high-level design, however, requires more elaborated means.

Various modelling guidelines for control algorithms are to be found in the industry standard [26] by MathWorks Automotive Advisory Board (MAAB). These directives give developers of automotive control systems a consistent notation, which can be extended with project-specific rules and used to obtain reusable and easy to integrate designs. However, these substantial guidelines were found to be similar or (seemingly) contradictive [32]. Therefore, we approach the design process issue from a different angle and build measurement-supported modelling strategy.

Numerous articles about software quality in Simulink developments mostly consider generic qualities or performance requirements, which should be preserved by the system and are defined by the stakeholders or developers. In case of Ford Motor Company or DEQX there were certain requirements regarding the sound quality [33] [34]. Simulink as a development environment is also described in perspective of process quality improvement, like in case of Toyota [35] and Lear [36]. We also intend to improve development process, but through control, assessment and improvement of the quality of the high-level design.

6. CONCLUSIONS AND FUTURE WORK DIRECTIONS

The software nowadays is growing rapidly in size and complexity. Nevertheless, it is expected to fulfil a general demand for high quality. Software quality control at the high-level design stage benefits in cost-effective quality management, since it is rather straightforward to introduce changes or fixes already at this point. Moreover, an

evaluation process is enabled, which facilitates making early and well-timed project decisions.

Our measurement approach is Simulink specific, applicable to the hierarchical model based design of all Simulink diagram levels. It can be considered at two design granularities, i.e. for individual layer of the Simulink model or particular (subsystem) block. We focused on the detailed assessment of the complexity within the layers of the model by considering not only structural, but also data complexity. Moreover, we concentrated on the evaluation of the fundamental features of the hierarchical design and interdependencies between the model elements. Therefore, instability and abstractness measurements were computed and the relation between these characteristics was further investigated. In order to illustrate how the metrics actually work, we supported our examination with the case study from the digital hydraulics domain.

We benefit from the measurement-supported modelling approach, as we can discover and expose possible sensitivities in the high-level design. The measurements presented are scalable, since they do not depend on the size of the model. Additionally, they are repeatable, as all Simulink designs should be under constant control in order to manage the quality of the design timely and efficiently. Moreover, they are inexpensive, because they are tool supported. Given that the measurements are applied at the early development phase, the identified problems are not propagated to later stages. This leads to better quality of the constructed system, with potentially fewer errors; moreover, it is cost-effective. The metrics are rather straightforward and can be applied cross-domain in order to tackle structural issues.

The metrics presented in this paper should be treated as guidelines, since they are identified as simple measurement categories proposed for the Simulink setting for the purpose of system design improvement. They should be used in a reasonable way according to the chosen standard and project goals.

In our future work we plan to verify the improvement of the quality of a design of Simulink models. We want to use case studies from various areas in order to collect evidence demonstrating that the measurement supported modelling aids the improvement of the design quality regardless of the system domain. Our goal is to show with our metrics that the overall development process can be improved through improvement of the design quality. This can be achieved by continuous monitoring and evaluation of the system design, as well as a design process with established quality measurements. We want to promote measurements as an essential part of quality improvement strategy.

7. ACKNOWLEDGEMENTS

This work was done within EFFIMA program coordinated by FIMECC.

We would like to thank Marina Waldén for her constructive comments on the paper and Pontus Boström for his support regarding Simulink environment. We are also grateful to Matti Linjama, Mikko Huova and Mikko Heikkilä (Intelligent Hydraulics and Automation Department, Tampere University of Technology) for providing us with the case study.

References

[1] Whitty Robin, *Research in Specification Metrics*, IEEE Colloquium on Software Metrics, (2002), pp.2/1 - 2/2.

[2] Stein Cara, Etzkorn Letha, Utley Dawn, *Computing Software Metrics from Design Documents*, Proceedings of the 42nd annual Southeast Regional Conference. ACM, Huntsville, Alabama (2004)

[3] *Software Errors Cost U.S. Economy $59.5 Billion Annually*. National Institute of Standards and Technology, Gaithersburg (2002)

[4] McQuillan Jacqueline, *On the Application of Software Metrics to UML Models*, Workshops and Symposia at MoDELS 2006. Spinger-Verlag, Genoa, Italy (2006)

[5] Simulink - Simulation and Model-Based Design, http://www.mathworks.com/products/simulink/

[6] Back Ralph, *Software Construction by Stepwise Feature Introduction*, ZB 2002. Springer, Heidelberg (2002)

[7] Eeles Peter, *Layering Strategies*. Rational Software White Paper (2002)

[8] Boström Pontus, Pląska Marta, Houva Mikko, Linjama Matti, Heikkilä Mikko, Sere Kaisa, Waldén Marina, *Contract-based Design in Controller Development and its Evaluation*, NODES 09: NOrdic workshop and doctoral symposium on DEpendability and Security. Linköping University Electronic Press, Linköping (2009)

[9] Boström Pontus, Morel Lionel, Waldén Marina, *Stepwise Development of Simulink Models Using the Refinement Calculus Framework.*, Theoretical Aspects of Computing (ICTAC2007). LNCS Springer, Macao (2007)

[10] Lew Ken, Dillon Tharam, Forward Kevin, *Software Complexity and its Impact on Software Reliability*, IEEE Transactions on Software Engineering, 14 (1988), pp.1645 - 1655.

[11] Banker Rajiv, Datar Srikant, Kemerer Chris, Zweig Dani, *Software Complexity and Maintenance Costs*, Communications of ACM, 36 (1993), pp.81-94.

[12] Pląska Marta, Huova Mikko, Waldén Marina, Sere Kaisa, Linjama Matti, *Quality Analysis of Simulink Models*, 12th International Conference on Quality Engineering in Software Technology, CONQUEST2009. dpunkt.verlag, Nuremberg (2009)

[13] Huova M., Pląska M., Siivonen L., Linjama M., Waldén M., Vilenius M., Sere K., *Controller Design of Digital Hydraulic Flow Control Valve* , The 11th Scandinavian International Conference on Fluid Power, SICFP'09. , Linköping, Sweden (2009)

[14] Pląska Marta, Waldén Marina, *Quality Comparison and Evaluation of Digital Hydraulic Control Systems*. Turku Center for Computer Science (TUCS), Turku (2007)

[15] Boström Pontus, Grönblom Richard, Huotari Tatu, Wiik Jonatan, *An Approach to Contract-Based Verification of Simulink Models*. TUCS, Turku (2010)

[16] Chidamber Shyam, Kemerer Chris, *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, 20 (1994), pp.476-493.

[17] McCabe Thomas, *A complexity measure*, IEEE Transactions on Software Engineering, Se-2, No. 4 (1976), pp.308-320.

[18] Concas Giulio, Marchesi Michele, Pinna Sandro, Serra Nicola, *Power-Laws in a Large Object-Oriented Software System*, IEEE Transactions on Software Engineering, 33 (2007), pp.687-708.

[19] Subramanyam Ramanath, Krishnan M., *Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects*, IEEE Transactions on Software Engineering, 29 (2003), pp.297-310.

[20] Gyimothy Tibor, Ferenc Rudolf, Siket Istvan, *Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction*, IEEE Transactions on Software Engineering, 31 (897-910), pp.2005.

[21] Hatton Les, *Reexamining the Fault Density-Component Size Connection*, IEEE Software, (1997), pp.89-97.

[22] Piwowarski Paul, *A nesting level complexity measure*, ACM SIGPLAN Notices, 17 (1982), pp.44 - 50.

[23] Martin R.C., *OO Design Quality Metrics. An Analysis of Dependencies*. (1994)

[24] Menkhaus Guido, Frei Urs, Wuthrich Jorg, *Analysis and Verigication of the Interaction Model in Software Design*, 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05). IEEE Computer Society, Shanghai (2005)

[25] Menkhaus Guido, Andrich Brigitte, *Metric Suite for Directing The Failure Mode Analysis of Embedded Software Systems*, 7th International Conference on Enterprise Information Systems (ICEIS). , Miami (2005)

[26] (MAAB) MathWorks, *Control Algorithm Modeling Guidelines Using Matlab®, Simulink®, and Stateflow®, Version 2.0*. (2007).

[27] Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B., Wesslén A., *Experimentation in Software Engineering: An Introduction*. Springer, Heidelberg (2000), pp.401-404.

[28] sldiagnostics, http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/slref/sldiagnostics.html

[29] Stateflow, http://www.mathworks.com/products/stateflow/

[30] MATLAB Central, http://www.mathworks.com/matlabcentral/fileexchange/5574

[31] Hosagrahara Arvind, Paul Smith P., *Measuring Productivity and Quality in Model-Based Design*. The MathWorks, Inc. (2006)

[32] Farkas Tibor, Hein Christian, Ritter Tom, *Automatic Evaluation of Modelling Rules and Design Guidelines*, From code centric to model centric software engineering: Practices, Implications and ROI.. Centre for Telematics and Information Technology, Enschede (2009)

[33] MathWorks - User Stories - DEQX, http://www.mathworks.com/products/signal/userstories.html?file=45581

[34] MathWorks - User Stories - Ford, http://www.mathworks.com/test-measurement/userstories.html?file=45578

[35] MathWorks - User Stories - Toyota, http://www.mathworks.com/products/simulink/userstories.html?file=45479&title=The MathWorks Tools Help Toyota Design for the Future

[36] MathWorks - User Stories - Lear, http://www.mathworks.com/company/user_stories/userstory45642.html

Publication 6

SMARTER Metrics

Marta Olszewska (Pląska)

Accepted to the 5th World Congress on Software Quality 2011, Shanghai, China, October-November 2011.

©2011 Shanghai Software Industry Association (SSIA). Reprinted with kind permission of SSIA.

5th World Congress for Software Quality – Shanghai, China – November 2011

SMARTER Metrics

Marta Olszewska (Pląska)
Åbo Akademi University
Turku Centre for Computer Science (TUCS)
Turku, Finland
Marta.Plaska@abo.fi

Abstract

This paper presents our vision on the quality metrics and measurements and their current position in the development of software systems. Our primary objective is to define the idea of SMARTER metrics - a collection of attributes, which we identified as key characteristics of metrics. Metrics with these desirable features are the driving forces in the mission of collecting evidence on the quality of the system. We characterise SMARTER as "more than SMART", that is Specific, Measurable, Actionable, Relevant and Timely, but also Easy to manage and Reusable. We emphasize the importance of quality control of the project via the suitable metrics and measurements. Our aim is to create a repository of SMARTER metrics, which consists of building blocks that can be (re)used when assessing the quality or forming a measurement plan.

The SMARTER framework is supported with an example of set of metrics for evaluating the quality of the design of Simulink models. These metrics have been established in our previous work and extensively impacted the SMARTER initiative.

1. Introduction

Quality of software systems depends on increasing scientific and technical knowledge. This regards both the practice of software development and empirical research in software technology. The quality measurements are the key factors building knowledge for software methodologies that are applied, since they provide real-life feedback for the scientific theory behind the development approach. Moreover, they are vital for software engineering activities, as they shape know-how of the software development itself through the empirical investigation, which regards practical aspects of the approach and feasibility of technology. Software system metrics are considered as indicators and guidelines towards the success of the project, instead of being the ultimate truth [1]. They are more and more regarded in terms of good practice and inevitable part of the development life cycle, rather than necessary evil.

Our goal is to manage the quality of the development by controlling the project at its every stage, regardless of the type of the development process (e.g. agile, waterfall, iterative). We want to monitor the development via the measurements and thus facilitate the documentation regarding the development. We consider the collection of evidence as a driving force when deciding about the development strategy, fine-tuning the methodology or introducing possible tactics to the process. Moreover, we believe that metrics and measurements are the way towards objective assessment of the quality of the systems, provided that the metrics are meaningful [2], and analyses and interpretation are done carefully. Furthermore, the measures are the essential factors towards quality improvement through assessment and comparison.

The idea of SMART metrics has been described in [3] [4], however it has not been heavily applied since then. The abbreviation stands for Specific, Measurable, Actionable, Relevant and Timely, and has been used in perspective of business success and marketing. Our contribution in this paper is the extension of SMART initiative to the SMARTER metrics framework, since we expect some more characteristics to be accomplished, that is for the metrics to be Easy to manage and Reusable (see Figure 1). Moreover, we apply metrics in the software systems setting. The novelty of this framework comes in employing it for the purpose of arranging the repository of metrics, where each metric would be suitable for the assessment of certain characteristic, but at the same time abstract enough to be adjusted or extended to a different development environment. This "database of metrics" consists of building blocks, which can be obtained whenever needed, fine-tuned and applied to the current setting.

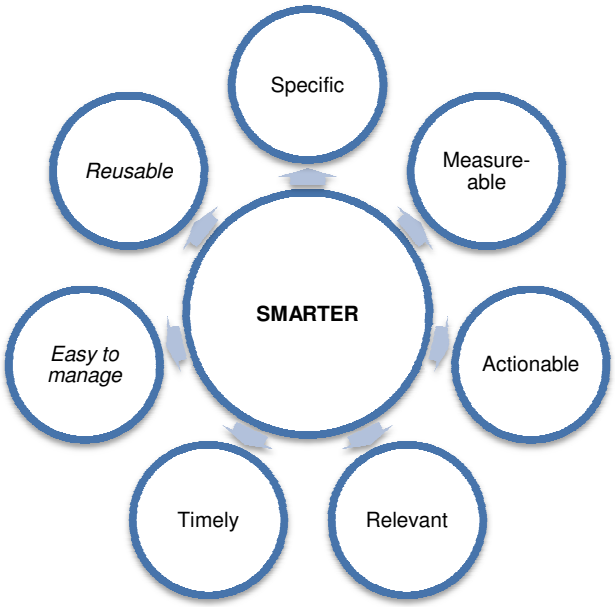


Figure 1. SMARTER attributes

The proposed approach to intelligent measurement elevates awareness of the development team about the status of the project. The "off-the-shelf" metrics are easy to access, choose, adjust, consolidate and apply at any time. It is beneficial for the developers to be able to assess the quality of their development or particular quality attribute of interest. Furthermore, it is additional gain for the managerial staff, since they can control the development through straightforwardly attainable measurements. SMARTER approach to metrics reduces the risk of collecting many unnecessary or perhaps meaningless measures.

2. SMARTER Approach to Quality

Our main motivation is to disseminate the idea of SMARTER approach to measurements as an integral element in software development process. We want the SMARTER metrics to be present during the entire lifecycle of systems, as part of the quality assurance activities, no matter what type of the system is under the development, e.g. embedded system, control system, web application, etc. Furthermore, we want to maintain the simplicity in metrics, as well as accomplish the simplicity of measuring activity through metrics.

There is a tendency of shifting the focus of measurements and quality control from end-stage to earlier development stages [5]. Making modifications and fixing defects in the modelling phase is much more cost and effort efficient, than performing the changes later, e.g. during the testing or deployment [6]. Moreover, the monitoring and management of systems nowadays indicates control throughout the duration of the project, whether it is dictated by standards, stakeholders or internally by the quality managers. It is founded on the experience within software systems development, where software creation and maintenance is regarded as an intrinsic part of business.

Although there are many Object-Oriented Metrics [7] [8], multiple code metrics that are language specific and tool-supported [9], there is a lack of metrics for e.g. (formal and semi-formal) specifications and formal modelling languages. Therefore there is a need for these metrics, since even if their existence and application does not bring immediate and direct profits money-wise, it may prevent from financial losses later on.

2.1 SMART

The approach of having SMART metrics was first mentioned in white paper [3] by Dave Trimble in 1996. His objective was to be able to have effective metrics as foundations for the evaluation and improvement of the business aspects in the projects. The group of metrics he presented falls into the financial and personnel

performance, as well as diagnostic categories. There are two levels considered: higher-level metrics for evaluation of performance with respect to e.g. business requirements, namely what is measured in the business process; and internal diagnostic metrics illustrating why the business process is not good enough.

SMART formulates strategy towards effective metrics and identifies the desirable characteristics of these. The metrics should be *Specific* in a sense of being clear i.e. explicit and directed at the subject under measurement. They should be *Measurable*, denoting that the data to be collected are complete and accurate. The understandability of metrics and acting upon the observed measurement results is expressed by feature *Actionable*. Assembling metrics that are meaningful and limited only to the important ones is illustrated as *Relevant*. *Timely* characteristic means data collection upon request, whenever a measurement is needed. These features combined enable the dynamic tracking of the development progress with a set of metrics that target significant artefacts.

2.2 E - Easy to manage

We extend the idea of SMART metrics with attribute *Easy to manage*. By that we signify that metrics should be aligned with the development process chain, in order to be able to support the management of project with metrics. This implies a large degree of automation, which should be incorporated into the tool chain as early as possible. Although some initial effort and costs are involved, the automation pays off later. Having the metrics integrated allows for the easier execution of corrective actions, since tuning the measurement criteria and setting the thresholds in the properties of the tool is much less demanding.

A good metric is the one that can be controlled, since only then it works well. A collection of metrics can be managed only if they are well documented and restricted in number; thus they minimise the costs even further and do not distract the personnel responsible for measurements. Therefore, a set of metrics should be carefully chosen with respect to the measurement criteria.

Metrics should be well defined and a record of their objectives should be unambiguous. They should remain practical, in order for their sustainability to be ensured. The primitives or the building blocks of the metrics, as well as relations between them should be clearly described, so that they can possibly be reused in a different setting.

2.3 R - Reusable

Metrics nowadays should be flexible enough to tackle global development trends. We have noticed a trend of *Reusing* the ideas from other domains and adjusting them to software field. This occurred e.g. with the agile development process, which originates from the automotive domain. Furthermore, metrics are now driven by the emerging areas and new applications of methodologies. Therefore, they can benefit from reuse of concepts known as beneficial and practised in other fields, e.g. engineering practices, in relation to which software is said to be 20 years behind [10]. This technology transfer should be dynamic, i.e. adaptable to requirements of constantly changing software.

Identifying new directions for metrics indicates the reuse of old and generic ideas, and locating them in the new setting. The cross-domain application of metrics signifies the process of their refinement, i.e. defining new meaning for the primitives or building blocks, extending the existing concepts and possibly adding factors to the formulas. Such an approach to metrics evolution is economical and we believe it has a significant potential.

A common downfall of quality measurements is the careless and inaccurate use of metrics, which can bring more damage than having no metrics at all. Hence, metrics should be verified in order to assure their validity. Reuse of metrics needs to be done critically, so that the measurements conform to the perception of the professionals within the domain under investigation. This can be achieved by practical validation [11] in a form of case studies or surveys, accompanied by expert opinion. In general, one should strive for the meaningful metrics using the metrics knowledge and common sense. Metrics that are useful in one area might not be reusable in other. Therefore, there is a motivation of having a repository of metrics created with SMARTER framework, which to a large degree would facilitate the process of selecting the suitable metrics.

3. Being Smart is Doing SMARTER

In our previous work we have created metrics and measurement programs for the assessment of rigorous types of developments, which were benefiting from Contract-Based Design approach [12] or Event-B formal method [13]. These measurements were applied for large and complex critical systems, such as digital hydraulic control valve system or highly critical industrial system within the space sector. We have noticed that there are commonalities in the process of creation of metrics and some of the characteristics of these metrics overlap.

In order to give background information on our way towards the SMARTER framework, here we shortly describe the subjects in our "metrics journey". Firstly, we built the syntax-based specification metrics for Event-B rigorous developments [14] by adjusting the well-known Halstead Software Science metrics [15]. We also investigated Simulink [16] modelling environment and used the Card and Glass complexity model [17], which was modified to Simulink setting [18]. Furthermore, the set of metrics available for Object-Oriented designs (OOD) founded by Robert C. Martin [19] was adapted to work in the Simulink environment [20]. All of these metrics were created in order to control the complexity of the system under construction and thus improve the maintainability and understandability aspects of quality. The processes of establishing these metrics, although specific for certain development settings, share some generic objectives. They encouraged the idea of initiating SMARTER measurements.

3.1 Example Metrics

In this paper we present the metrics established in [20] to show how the SMARTER approach works in practise. We investigate measures associated with model hierarchy (nesting) and relations between the system layers. We examine these design characteristics, since we regard them as *Relevant* - they impact the understanding of the design, size and the overall complexity of the system [21]. We were inspired by the metrics created for OOD. Robert C. Martin [19] established several package metrics evaluating the conformance of a design to a pattern of dependency and abstraction. Our metric application setting is Simulink - a powerful high-level graphical modelling tool, which is highly appreciated in industry, mainly in scientific and engineering areas, and used for the purpose of model-based design and multi-domain simulation.

Since we want to be able to dynamically (i.e. *Actionable* and *Timely*) assess the cross-layer relationships between the Simulink structures, we introduce the concept of coupling measures to the Simulink model design. The *Measures* are based on the dependencies between abstract and concrete units in the model, i.e. subsystem and functional blocks. In our research we analyse the structure of Simulink-S*pecific* model by investigating two characteristics of the design: instability and abstractness. Here we *Reuse* the idea that has been successfully used in OOD; however, we adjust and extend it to Simulink models. For our setting we redefine the primitives and specify the meaning of the relations between them.

We tackle Simulink block diagrams which consist of layers (Figure 2), where each layer is defined by subsystem blocks (*mx+b*) or functional blocks (*Slope*, *Sum*, *Intercept*). The subsystem blocks can be treated as an abstraction of several blocks and are present for the purpose of facilitating the structuring of a diagram, as shown in Figure 3. They have no impact on the meaning of the diagram and do not define a separate one.

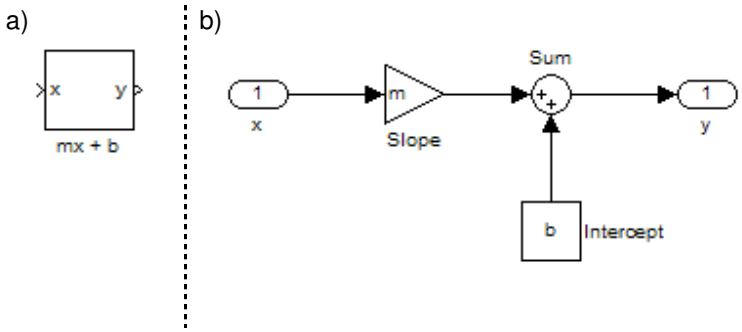


Figure 2. Sample Simulink diagram: a) subsystem block, b) its contents with functional blocks [22]

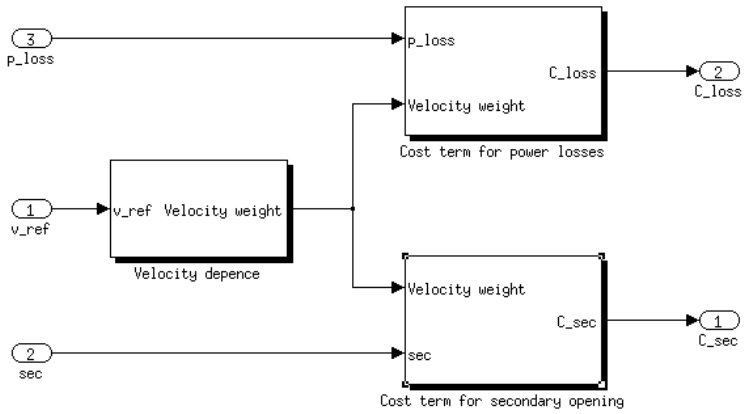


Figure 3. Sample Simulink diagram – one layer with three subsystem blocks [23]

Subsystem and functional blocks depend on each other within one layer through incoming and outgoing signals (in Figure 2 and 3 these are the links with arrows). We define *afferent coupling between blocks* (*CaB*) as the number of the destination blocks for the block under analysis. *Efferent coupling between blocks* (*CeB*) is defined as the number of source blocks for the given block. In both measures each dependent block is counted only once, in spite of the number of signals linking it to a given block. The count is equal to zero if the block is not connected to other blocks by incoming or outgoing signals.

Higher number of couplings between blocks signifies that the blocks are more dependent, which means that they rely on the outcome of other blocks. These blocks can also be responsible, i.e. they are heavily dependent upon. Both of these features lead to the so called ripple effect, which means that introducing any change to the block has a considerable effect on other blocks, forcing additional changes. Moreover, high number of interactions stands against modularity and reuse, as well as augments the maintenance costs. Additionally, high coupling points towards high complexity and thus increases the costs of testing. It also denotes that the design requires meticulous analysis and inspections, as it is troublesome to understand the existing dependencies. These observations comply with the generic perception known in OOD.

Instability I of a (subsystem) block is defined as the number of efferent couplings between blocks divided by the sum of afferent and efferent couplings between blocks, which is given by the equation:

$$I = \frac{CeB}{CeB+CaB},$$

where *CeB* and *CaB* are defined as before. The values of this metric range from zero (no incoming signals) to one (only incoming signals), where $I = 0$ denotes a completely stable (subsystem) block, and $I = 1$ signifies a maximally instable (subsystem) block.

Since instability is a characteristic defined by coupling measures, it impacts respective qualities of the system, i.e. modifiability, reusability and understandability. It is a sign for the developers to do the inspection of a given part of the design and possibly make changes. The (subsystem) blocks that are stable (I=0) have two features: they are simultaneously independent and responsible. Same rule has been applied in OOD, where interfaces are those entities that are fully abstract and, as such, should be stable and not altered.

We define *abstractness A* of a block as a ratio of the number of contained abstract (i.e. subsystem) blocks (*NaB*) to the total number of blocks in a layer the given block represents (*NB*), which is given by formula:

$$A = \frac{NaB}{NB}.$$

The presented metric has the range <0..1>, where 0 denotes a concrete block and 1 represents a completely abstract block. In case the block is a leaf block, i.e. the block in the lowest level of Simulink model, we define the abstractness *A* to be zero, meaning that the block is concrete.

According to MathWorks Automotive Advisory Board (MAAB) style guidelines [24], every level of a model should be designed with blocks of the same type. This means that the diagram should be structured so that the subsystem blocks are not mixed in one layer with functional blocks. In principle, the abstractness rule

from OOD is advised to be used in Simulink model design, placing the subsystem blocks in one layer and basic blocks in a separate one.

Abstractness as a characteristic impacts the understandability and readability of the model, as well as its maintainability and reuse. Making a subsystem block more abstract signifies hiding some of its details that are unnecessary at this level of comprehension. This resembles the encapsulation mechanisms found in object-oriented programming and design.

As it already has been confirmed for OO methodologies, there is a relationship between stability and abstractness. We have indicated such a correlation to be true in the Simulink environment, as well [20]. In order to confirm our claim, we identify the *distance* metric *D*, which we represent graphically as a chart (Fig. 4), where we take instability *I* and abstractness *A* as horizontal and vertical axis respectively. *D* is computed as a normalised sum of these values decreased by 1, which is given by the formula:

$$D = |A + I - 1|.$$

The resulting values range from *D*=0 to *D*=1, where the desirable values oscillate around *D*=0. The *D* metric depicts the correlation between abstractness and stability for a (subsystem) block; it is an indicator of balance between these two characteristics. Therefore, when the distance exactly equals zero, it signifies that the balance between these two is optimal. According to the formula the desired situation is when the subsystem blocks are either totally stable and abstract (*I*=0 and *A*=1) or entirely instable and concrete (*I*=1 and *A*=0). In Figure 4 it is illustrated as a solid line, called “The Main Sequence” in OOD.

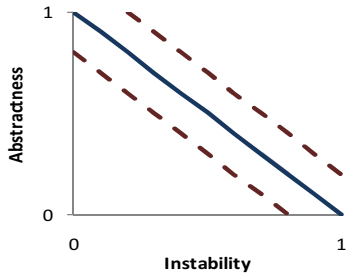


Figure 4. Graph representation of the distance metric *D*

Since when modelling in Simulink we want to follow the guidelines concerning the abstraction given by MAAB, the value *D*=0 will not be an optimal one. We have to set a margin fluctuating around the value proposed by Martin [19], to truly indicate that a (subsystem) block is coincident with the optimal balance with respect to its stability and abstractness. We called this a “zone of acceptance”. Its value should be project-specific. All the blocks that have a value of *D* close to The Main Sequence (between dashed lines) are said to have a good design. On the other hand we can safely state that the value *D*=1 means that the design is far from the desired characteristics and it should be carefully reviewed, and possibly restructured.

On the graph these problematic areas are the lower-left quadrant of the chart called the “zone of pain” and upper-right quadrant named “zone of uselessness”. The former stands for the case when the (subsystem) block is very stable, i.e. has a lot of dependencies, but it's not extensible, i.e. has no abstract subsystem blocks. The latter represents a situation when a block is abstract, very extensible, but nothing depends on it, i.e. it is not used. This information highlights the priorities for the design inspections and potential design modifications. It is also consistent with the principles of OOD presented in [19].

3.2 SMARTER Insights

The presented set of metrics is S*pecific* to the Simulink modelling environment. It is tool supported with the metrics plug-in, an extension of the Simulink verification tool [25]. The measurements can thus be automatically and dynamically collected during the modelling process. This makes them *Timely* and *Measurable*, as well as repeatable and cost-effective. The metrics themselves are straightforward enough to be appreciated by the developers. Moreover, the measurements are easy to understand to be followed and enable constant control in the early development stage for the managerial staff.

The measurements can be used at the modelling stage of the development in order to detect and remove possible defects in the design. The project staff benefits form these metrics, since they are *Actionable*. Constant control over the model and early application of quality measurements limits or even prevents the propagation of design issues. Managing maintainability and understandability through measuring abstractness and instability characteristics in the early phase is important, since it positively impacts the financial side of the project. Therefore the metrics are considered as *Relevant*.

The presented metrics for Simulink were inspired by the well-known OOD metrics. The main ideas given in [19] are abstracted away, *Reused*, extended and adjusted to the specificity of Simulink setting. They are *Easy to manage*, since the primitives and relations between the characteristics are well defined and simple. Moreover, the data collection is mechanised, therefore the measurement reports can be obtained at any time.

4. Conclusions and Future Work Directions

In this paper we presented the SMARTER metrics framework characterising desirable attributes of metrics. Our motivation is to establish a repository of metrics preserving these attributes in a form of off-the-shelf metrics. Such database would facilitate measurement activities at any point of the software system development.

We are aware of attempts of creating a reusable repository specific for Web metrics [26], as well as of existing research on business process metrics [27]. However, these catalogues are restricted to the certain application field. We would like to create a generic repository that provides information on the quality attributes that it measures, as well as notification about the meaningfulness of a standard metric with respect to certain application domain and development setting. A compendium of essential software metrics data was compiled from the International Software Benchmarking Standards Group's repository of software projects [28], however it is in a form of a book, whereas we aim at a dynamic and interactive computer-based repository of metrics. We believe that SMARTER metrics can facilitate the software quality analysis and evaluation, as well as its management in organizations.

Acknowledgements

This work was done within EFFIMA (Energy and Life Cycle Cost Efficient Machines) program coordinated by FIMECC (Finnish Metals and Engineering Competence Cluster).

We would like to thank Marina Waldén for her supervision, Pontus Boström for the expertise regarding the Simulink environment and Jeanette Heidenberg for fruitful discussions on software quality, as well as Mikołaj Olszewski for the encouragement. Substantial gratitude to Matti Linjama, Mikko Huova and Mikko Heikkilä (Intelligent Hydraulics and Automation Department, Tampere University of Technology, Finland) for providing us with the Simulink case studies.

References

1. Kan S., *Metrics and Models in Software Quality Engineering*. Addison-Wesley (2003).

2. Fenton N., Pfleeger S., *Software Metrics. A Rigorous & Practical Approach*. PWS Publishing Company (1997).

3. Trimble D., *How to Measure Success: Uncovering The Secrets Of Effective Metrics*. ProSci Whitepaper (1996)

4. Phelps B., *Smart business metrics*. FT Prentice Hall (2004).

5. Whitty R., *Research in Specification Metrics*, IEEE Colloquium on Software Metrics, (2002), pp.2/1 - 2/2.

6. *Software Errors Cost U.S. Economy $59.5 Billion Annually*. National Institute of Standards and Technology, Gaithersburg (2002)

7. Chidamber S., Kemerer C., *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, 20 (1994), pp.476-493.

8. STAN - Structure Analysis for Java, stan4j.com

9. JavaNCSS - A Source Measurement Suite for Java, http://www.kclee.de/clemens/java/javancss/index.html. Last accessed 20th July 2011

10. Aoyama M., *Co-Evolutionary Service-Oriented Model of Technology Transfer in Software Engineering*, Workshop on Technology Transfer for Software Engineering, ICSE 2006. ACM, Shanghai (2006)

11. Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B., Wesslén A., *Experimentation in Software Engineering: An Introduction*. Springer, Heidelberg (2000), pp.401-404.

12. Boström P., Pląska M., Houva M., Linjama M., Heikkilä M., Sere K., Waldén M., *Contract-based Design in Controller Development and its Evaluation*, NODES 09: NOrdic workshop and doctoral symposium on DEpendability and Security. Linköping University Electronic Press, Linköping (2009)

13. Event-B.org, http://www.event-b.org/index.html, Last accessed 20th July 2011

14. Olszewska (Pląska) M., Sere K., *Specification Metrics for Event-B Developments*, 13th International CONference on QUality Engineering in Software Technology. iSQI, Dresden (2010)

15. Halstead M.H., *Elements of Software Science*. Elsevier North Holland (1977), pp.128.

16. Simulink - Simulation and Model-Based Design, http://www.mathworks.com/products/simulink/. Last accessed 20th July 2011

17. Card D., Glass R., *Measuring Software Design Quality*. Prentice Hall (1990), pp.144.

18. Pląska M., Huova M., Waldén M., Sere K., Linjama M., *Quality Analysis of Simulink Models*, 12th International Conference on Quality Engineering in Software Technology, CONQUEST2009. dpunkt.verlag, Nuremberg (2009)

19. Martin R.C., *OO Design Quality Metrics. An Analysis of Dependencies*. (1994)

20. Olszewska (Pląska) M., *Simulink-Specific Design Quality Metrics*. Turku Centre for Computer Science (TUCS), Turku (2011)

21. Piwowarski P., *A nesting level complexity measure*, ACM SIGPLAN Notices, 17 (1982), pp.44 - 50.

22. MathWorks, http://www.mathworks.com/help/toolbox/simulink/ug/bsp2zlg-1.html. Last accessed 20th July 2011

23. Huova M., Pląska M., Siivonen L., Linjama M., Waldén M., Vilenius M., Sere K., *Controller Design of Digital Hydraulic Flow Control Valve* , The 11th Scandinavian International Conference on Fluid Power, SICFP'09. , Linköping, Sweden (2009)

24. MathWorks Automotive Advisory Board MAAB, *Control Algorithm Modeling Guidelines Using Matlab®, Simulink®, and Stateflow®, Version 2.0*. (2007).

25. Boström P., Grönblom R., Huotari T., Wiik J., *An Approach to Contract-Based Verification of Simulink Models*. TUCS, Turku (2010)

26. Olsina L., Lafuente G., Pastor O., *Towards a Reusable Repository for Web Metrics*, Journal of Web Engineering, Vol.1 No.1 (2002), pp.61-73.

27. Hong H., GuiFang Y., QinBao S., KeGa H., *The Research of Metrics Repository for Business Process Metrics*, 2nd International Symposium on Computational Intelligence and Design. IEEE Computer Society, Changsha (2009)

28. ISBSG , *The Software Metrics Compendium*. International Software Benchmarking Standards Group (2002), pp.125.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2671-7
ISSN 1239-1883