Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into
• seminar
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: http://www.gi-ev.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-273-4

This LNI Volume, 179, contains the Proceedings of FM+AM`2010: Second International Workshop on Formal Methods and Agile Methods which took place in Pisa (Italy) on the 17th of September 2010 under the umbrella of the 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM`2010. This book contains 1 invited paper and 4 reviewed papers on the following topics: • Are Formal Methods Ready for Agility? • Agile Formality – A Mole of Software Engineering Practices • State-based Coverage Analysis and UML-driven Equivalence Checking • Improved Under-Specification for Model-based Testing in Agile Development • Formal Analysis of High-level Graphical SOA Design.

S. Gruner, B. Rumpe (Eds.): FM+AM`2010 – Formal and Agile Methods

179

GI-Edition

Lecture Notes in Informatics

Stefan Gruner, Bernhard Rumpe (Eds.)

FM+AM`2010

Second International Workshop on Formal Methods and Agile Methods

Pisa (Italy), 17 September 2010

Proceedings

Stefan Gruner, Bernhard Rumpe (Eds.)

FM+AM`2010

Second International Workshop on Formal Methods and Agile Methods

17 September 2010
Pisa (Italy)

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-179

ISBN 978-3-88579-273-4
ISSN 1617-5468

Volume Editors
Stefan Gruner
Department of Computer Science
Universiteit van Pretoria
0002 Pretoria, South Africa
Email: sgruner@cs.up.ac.za
Bernhard Rumpe
Lehrstuhl Informatik III – Software Engineering
Rheinisch-Westfälische Technische Hochschule Aachen
Ahornstraβe 55
52074 Aachen, Germany
Email: rumpe@se-rwth.de

printed by Köllen Druck+Verlag GmbH, Bonn

2nd Workshop on Formal Methods and Agile Methods FM+AM'2010: Foreword and Editorial Preface

Stefan Gruner[1], Bernhard Rumpe[2]

[1] Department of Computer Science, University of Pretoria, South Africa
sgruner@cs.up.ac.za

[2] Software Engineering, RWTH Aachen, Germany, http://se-rwth.de/

Abstract: FM+AM'2010, the 2nd international workshop on Formal Methods and Agile Methods in software engineering, took place under the umbrella of the 8th IEEE international conference on Software Engineering and Formal Methods, SEFM'2010, in Pisa (I), September the 17th, 2010. This workshop had one invited lecture, and four reviewed papers presented. This editorial preface motivates the main ideas that were guiding this workshop and provides some information about its committee as well as its paper review and selection procedure.

1 Motivation

Formal Methods (FM) and Agile Methods (AM) are two rather different approaches to software design and development. Though in software engineering, FM and AM are widely regarded as incommensurable methodological antagonists [B+09], both have their offspring in the continuing struggle, formerly known as 'software crisis', for a high software product quality at the end of an efficient production process,. Though we do not speak of a 'software crisis' in these days any more, (and though also many hardware and civil engineering projects are running late, go over their budgets and deliver products of questionable quality), we still cannot be satisfied with the current ways of software development. Both FM and AM come as particular responses to these challenges of software software development, which are typically characterised by a too slow and error-prone software production process in combination with a too low software product quality at the end of the production process.

While FM have concentrated on techniques for high-quality results, they are perceptively slow and tedious and thus costly. They are assumed to be 'heavyweight', where every single step needs to be traceable, validated or even verified. AM on the other hand are concentrating on efficiency of the development process together with a number of techniques to focus on the right requirements, etc. AM are generally regarded as 'lightweight' and thus more amenable, when dealing with small projects. State of the art has it that AM are used in smaller projects and when risk does not involve substantial amount of money or even lives.

The goal of this workshop is to discuss these issues, and to bring these different viewpoints and aspects closer to each other. As mentioned above: whereas the FM colleagues in software engineering are most concerned about the product correctness aspect of software design, the AM colleagues seem to be most concerned about the aspect of production velocity, though this distinction between FM and AM is (of course) somewhat coarse and should be taken with a grain of salt. For both 'schools' of software engineering the often questioned 'engineering-ness' of software engineering, as an academic and practical discipline, is at stake [DR09]. This is also the reason why combinations of the two methodological approaches, FM and AM, should be feasible and possibly fruitful. This is the theme which this workshop, FM+AM'2010 (under the umbrella of SEFM'2010, Pisa, Italy, September 2010), was meant to explore, with the ultimate aim of making formally sound methods of software development faster, and rapid development methods more formally sound. This could –for example– be achieved by providing formally verified CASE tools to agile development groups, or by introducing agile work methods, such as working in pairs and short iterations, into the domain of formal model design. Already in the year 2004 the related idea of extreme modelling (in analogy to extreme programming) had been proposed [A+04]; a similar idea is nowadays called agile modelling.[1] More pragmatically it might also be helpful to use formal methods only in certain critical parts of an otherwise agile development project, whereby special care must be taken about the accurate identification and the precise definition of such critical parts.

However, not too many software engineering researchers seem to be interested in the combination of these themes these days. Similar to what has been the case at this workshop's predecessor, FM+AM'09 [Gr09] (under the umbrella of ICFEM'09, Rio de Janeiro, Brasil, December 2009), we have had –again– a rather small number of paper submissions; see below for details of our workshop's submission and reviewing phase. A related workshop, prior to FM+AM'09, was organised in the year 2008 by Meyer et al. [M+08].

[1] http://www.agilemodeling.com/

2 Call for Papers and Response

After the permission for this workshop FM+AM'2010 had been given by the committee of the SEFM'2010 conference (see acknowledgments below), a programme committee was assembled with experts in both fields, FM and AM. Thereafter, several calls for papers were widely distributed via a number of international mailing lists as well as the workshop's website on the internet. Following those calls, five papers were submitted in June 2010. After the review process, whereby each paper got at least four reviews by different members of the Programme Committee (see below), four papers were accepted for presentation at the workshop. Their revised versions (re-submitted after review) appear in this volume of the LNI. Moreover, *Peter Gorm Larsen* was invited to present a keynote lecture, and he kindly accepted our invitation. The contents of his lecture is also represented in this book, in the form of an invited paper (which was not reviewed by the workshop's Programme Committee).

3 Programme Committee and Additional Reviewers

The following experts (in alphabetical order by surnames) from both fields of Formal Methods and Agile Methods wrote the reviews and recommendations about the papers that had been initially submitted in June 2010:

- *Scott Ambler*, Ambysoft Inc. (Canada)
- *Robert Eschbach*, Fraunhofer Institute IESE (Germany)
- *Jaco Geldenhuys*, University of Stellenbosch (South Africa)
- *Stefania Gnesi*, National Italian Research Foundation ISTI-CNR (Italy)
- *Stefan Gruner* (FM+AM'2010 Workshop Chair and Proceedings Co-Editor), University of Pretoria (South Africa)
- *Horst Lichter*, RWTH Aachen (Germany)
- *Shaoyin Liu*, Hosei University (Japan)
- *Franco Mazzanti*, National Italian Research Foundation ISTI-CNR (Italy)
- *Pieter Mosterman*, McGill University (Canada)
- *Jürgen Münch*, Fraunhofer Institute IESE (Germany)
- *Kees Pronk*, Technical University of Delft (The Netherlands)
- *Bernhard Rumpe* (Proceedings Co-Editor), RWTH Aachen (Germany), assisted by *Christoph Herrmann* and *Antonio Navarro Pérez*

- *Holger Schlingloff*, Humboldt-University of Berlin (Germany)
- *Alberto Sillitti*, Free University of Bozen/Bolzano (Italy)
- *Willem Visser*, University of Stellenbosch (South Africa)
- *Xiaofeng Wang*, LERO Institute (Ireland), assisted by *Carlos Solis*

4 Acknowledgments

The editors of this volume would like to thank *Heinrich Mayr*, Editor-General of the LNI series, for his permission to publish these FM+AM'2010 workshop proceedings in this volume of the LNI. Many thanks to the organising committee of this workshop's umbrella conference, SEFM'2010, and there especially to *Maurice ter Beek*, Workshop Chair of SEFM'2010, for having made this workshop possible on the premises of the Italian national research council, ISTI-CNR, in Pisa (I). Thanks to *Peter Gorm Larsen* for having accepted our invitation and for having presented his invited lecture to the audience of our workshop. Thanks also to all authors and co-authors of the reviewed papers for their contributions. Thanks to all members of the Programme Committee, as well as to their additional reviewers and assistants, for their thoughtful comments and critique of the submitted papers during the review procedure. Last but not least thanks to *Cornelia Winter* (LNI Office), *Jürgen Kuck* (Köllen Printing and Publishing), *Agnes Koschmider*, and *Alexander Paar* for their help in the production process of this book.

Pretoria and Aachen, September 2010

References

[A+04] Augusto, J.; Howard, Y.; Gravell. A; Ferreira, C.; Gruner, S.; Leuschel, M.: *Model-based Approaches for validating Business-Critical Systems*. In STEP'04 Proceedings of the 11th Annual International Workshop in Software Technology and Engineering Practice, pp. 225-233, IEEE Computer Society Press, 2004.

[B+09] Black, S.; Boca, P.P; Bowen, J.P.; Gorman, J.; Hinchey, M.: *Formal versus Agile, Survival of the Fittest?* IEEE Computer, Vol. 42, No. 9, pp. 37-45. IEEE Computer Society Press, 2009.

[DR09] Denning, P.J.; Riehle, R.D.: *Is Software Engineering Engineering?* Communications of the ACM, Vol. 52, No. 3, pp. 24-26. ACM Press, March 2009.

[Gr09] Gruner, S. (ed.): *FM+AM'09 Workshop on Formal and Agile Methods – Editorial Preface and Foreword*. Innovations in Systems and Software Engineering, Vol. 6, pp. 135-136, Springer-Verlag, 2009 / 2010.

[M+08] Meyer, B.; Nawrocki, J.R.; Walter, B. (eds.*): IFIP Workshop on Balancing Agility and Formalism in Software Engineering*. Lecture Notes in Computer Science, Vol. 5082, Springer-Verlag, 2008.

Contents

Part I: Invited Lecture

Part II: Reviewed Papers

Part I

Invited Lecture

Are Formal Methods Ready for Agility?

A Reality Check

Are Formal Methods Ready for Agility? A Reality Check

Peter Gorm Larsen, Aarhus School of Engineering, Denmark, pgl@iha.dk
John Fitzgerald, Newcastle University, UK, John.Fitzgerald@ncl.ac.uk
Sune Wolff, Terma A/S, Denmark, sw@terma.com

Abstract: The integration of agile software development techniques with formal methods has attracted attention as a research topic. But what exactly is to be gained from attempting to combine two approaches which are seen as orthogonal or even opposing, and to what extent do formal methods already support the principles of agility? Based on the authors' experience in applying lightweight tool-supported formal methods in industrial projects, this paper assesses the readiness of formal methods technologies for supporting agile techniques and identified areas in which new research could improve the prospects of synergy between the two approaches in future.

1 Introduction

Formal methods are a response to the challenge of complexity in computer-based systems, and the defects that arise as a result. They are techniques used to model and analyse complex computer-based systems as mathematical entities. Producing a mathematically rigorous model of a complex system enables developers to verify or refute claims about the putative system at various stages in its development. Formal methods can be applied to models produced at any stage of a system's development, from high-level models of abstract requirements to models of the characteristics of running code, such as memory usage [WLBF09]. The motivations for including formal methods in software development are to minimise defects in the delivered system by identifying them as soon as they arise, and also to provide evidence of the verification of critical system elements. Formal methods are highly diverse, in part because of the variety of domains in which they have been applied. Notable applications have been in the areas of communications, operating system and driver verification, processor design, the power and transportation sectors.

In spite of their successful application in a variety of industry sectors, formal methods have been perceived as expensive, niche technology requiring highly capable engineers [Sai96]. The development of stronger and more automated formal analysis techniques in the last decade has led to renewed interest in the extent to which formal techniques can contribute to evolving software development practices.

The principles of agile software development emerged as a reaction to the perceived failure of more conventional methodologies to cope with the realities of software development in a volatile and competitive market. In contrast with some established development approaches, which had come to be seen as necessary fictions [PC86], agile methods were

characterised as incremental (small software releases on a rapid cycle), cooperative (emphasising close communication between customers and developers), straightforward to learn and modify, and adaptive to changes in the requirements or environment [ASRW02]. Four value statements[1] summarise the principles of the approach. Proponents of agile techniques value *Individuals and interactions* over processes and tools, *working software* over comprehensive documentation, *customer collaboration* over contract negotiation and *responding to change* over following a plan.

Agile methods have received considerable attention, but as Turk et al. have pointed out, they do appear to make some underlying assumptions [TFR02]. For example, close customer interaction assumes the ready availability of an authoritative customer; a lower value placed on documentation assumes that documentation and software models are not themselves first-class products of the process; the emphasis on adaptation to changing conditions assumes a level of experience among developers. Since not all projects satisfy these assumptions, it has been suggested that agile approaches are unsuited for distributed development environments, for developments that make extensive use of subcontracting or that require to develop reusable artifacts, that involve large teams or involve the development of critical, large or complex software.

Given the range of both formal and agile methods, their respective pros and cons, can the two work to mutual benefit, or is the underlying principle of rigorous model-based analysis incompatible with the rapid production of code and the favouring of code over documentation? This paper briefly examines some of the existing work on this question (Section 2). A review of the authors' experience in the focused "lightweight" application of the formal method VDM in industry (Section 3) is followed by a review of the four value statements of the agile manifesto (Sections 4). In each case, we ask whether formal methods as they are now are really able to help achieve the value goal, and what research might be needed to bridge the gaps between the two approaches. Section 5 provides concluding remarks.

2 Formal Methods and Agility

The relationship between formal and agile methods has been explored for more than five years. However the issues have recently been brought into focus by Black et al. in their article for IEEE Computer in 2009 [BBB+09]. Some researchers have sought to develop hybrid methods that benefit from both rigour and agility. For example, Ostroff et al. [OMP04] seek to harness Test-Driven Design, a well-known agile technique, with a more formal method of Design by Contract. Niu and Easterbrook [NE05] argue for the use of machine-assisted model checking to address the problem of partial knowledge during iterations of an agile process. López-Nores et al. [Mar06] observe that evolution steps in an agile development typically involve the acquisition of new information about the system to be constructed. This new information may represent a refinement, or may introduce an inconsistency to be resolved through a retrenchment. Solms and Loubser [SL10] describe a service-oriented analysis and design methodology in which requirements are specified as

[1] `http://agilemanifesto.org/`

formal service contracts, facilitating auto-generation of algorithm-independent tests and documentation. Model structure is formally defined and can be verified through a model validation suite. del Bianco et al. [dBSK10] weave agile and formal elements together into an incremental development process.

Some work seeks to develop more agile formal methods. For example, Eleftherakis and Cowling [EC03] propose XFun, a lightweight formal method based on the use of X-machines and directed at the development of component-based reactive systems. Suhaib et al. [SMSB05] propose an iterative approach to the construction of formal reference models. On each iteration, a model is extended by user stories and subjective to regressive verification. Liu [Liu09] suggests that Structured Object-Oriented Formal Language is a viable tool for use in an agile process.

But is there really a need to combine formal and agile techniques? Certain product types call for an integration of agile and formal techniques in their development, particularly where the need to respond to competitors in a lively market leads to requirements volatility, or where the characteristics of underlying components change become available. The embedded systems market is one field with both of these characteristics[2]. Indeed our current project DESTECS[3] addresses the need to improve the rate at which engineers from different disciplines iterate through early design stages.

For the developer wishing to combine agile and formal practices, it is wrong to think that the term "formal methods" refers to a single development process whether based on refinement or on post-factor verification. Similarly, it is inappropriate to think of agile software development as a set of practices that must be adopted wholesale. Both agile and formal techniques are just that – sets of techniques that should be combined to suit the needs of the product and the character of the development team. The developer is not, however, helped much by the existing literature. It is not always clear whether researchers aim to promote the use of formal methods by showing that they can be added to agile processes, or whether the aim is to produce more agile formal methods to be applied in the usual domains. Perhaps most importantly, there are few empirical results on which to base methods decisions.

3 Experience with VDM

The Vienna Development Method (VDM)[4] is a well established set of techniques for the construction and analysis of system models. VDM models in their most basic form consist of type and value definitions, persistent state variables, pure auxiliary functions and state-modifying operations. Data abstraction is supported by abstract base types such as arbitrary reals and finite but unconstrained collections (sets, sequences and mappings) [FLV08]. Data types can be restricted by arbitrary predicates as invariants. Functional abstraction is supported by implicit pre/post specification of both functions and op-

[2] We note that del Bianco et al. [dBSK10] use an example from this domain.

[3] `www.destecs.org`

[4] `www.vdmportal.org`

erations. The basic VDM language has been extended with facilities to support object-orientation, concurrency, real-time and distribution [FLM+05].

VDM's rich collection of modelling abstractions has emerged because industrial application of the method has emphasised modelling over analysis [FL07]. Although a well worked-out proof theory for VDM exists [BFL+94], more effort has gone into developing tools that are truly "industry strength". The VDMTools system, originated at IFAD A/S in Denmark, and now further developed and maintained by CSK Systems in Japan, supports the construction, static analysis and type checking of models, generation of proof obligations and, above all, highly efficient testing and debugging [FLS08]. These facilities have been essential to VDM's successful industry application in recent years. The same is true of Overture[5] [LBF+10], the new community tools for VDM, in which the development has been in part driven by a desire to interface VDM models of discrete event systems with quite heterogeneous models from other engineering disciplines, such as continuous time models of controlled plant [BLV+10]. Although research on proof support has been ongoing for some years [CJM91, DCN+00, VHL10], the tools have never fully incorporated proof, mainly due to a combination of lack of demand from industry users, and a lack of robustness and ease of use on the part of the available tools.

The majority of the industrial applications of VDM can be characterised as uses of "lightweight" formal methods in the sense of Jackons and Wing [JW96]. Early experience of this was gained in the 1990s when the ConForm project compared the industrial development of a small message processing system using structured methods with a parallel development of the same system using structured methods augmented by VDM, supported by VDMTools [LFB96]. A deliberate requirements change was introduced during the development in order to assess the cost of model maintenance, and records were kept of the queries raised against requirements by the two development teams. The results suggested that the use of a formal modelling language naturally made requirements more volatile in the sense that the detection of ambiguity and incompleteness leads to substantial revision of requirements. To that extent, advocates of formal methods have to embrace the volatility of requirements by the very nature of the process that they advocate.

The use of a formal method to validate and improve requirements has been a common theme in VDM's industrial applications [LDV97, SL99, PT99]. For example, two recent applications in Japanese industry, the TradeOne system and the FeliCa contactless chip firmware [KCN08, KN09], have used formal models primarily to get requirements right. In some situations, such as the FeliCa case, the formal model itself is *not* primarily a form of documentation – it is merely a tool for improving less formally expressed requirements and design documents that are passed on to other developers. In these applications, the trade-off between effort and reward of proof-based analysis has come down against proof, but in favour of a carefully scoped use of the formalism.

Several VDM industry applications have made successful use of high volume testing rather than symbolic analysis as a means of validating models. So for example the FeliCa chip was tested with 7000 black box tests and around 100 million random tests all with both the executable VDM specification as well as with the final implementation in C. The in-

[5] www.overturetool.org

dustrial VDM applications mentioned here can be characterised as processing aspects that are similar to the approach taken in an agile software development. Throughout there has been a continued focus on the needs of the customer and the process of applying VDM has been one for removing more and more uncertainty rather than focusing on obtaining a perfect system by verification. So, although we have never described our approach as "agile" it addresses some agile development principles.

4 The Agile Manifesto Meets Formal Methods

The four value statements of the agile manifesto are supported by 12 principles[6]. In this section we consider each value statement and the principles that relate to it. For each value statement, we review the extent to which formal methods support it today, discuss some deficiencies and suggest future research to remedy these.

4.1 Individuals and Interactions over Processes and Tools

This value statement emphasises the technical competence and collaboration skills of the people working on the project and how they work together. If this is not considered carefully enough, the best tools and processes will be of little use[7]. Two of the 12 principles supporting the agile manifesto are relevant:

- Build projects around motivated individuals. Give them the environment and support their need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

The most important resources available to a project are the people working on it and not the tools or methods they use. But once the right people have been chosen, neither the tools nor the processes should be disregarded.

A criticism of our work on ConForm, as of other demonstrations of the effectiveness of formal modelling, is that we employed clever "pioneering" people to do the formal methods work and they were bound to do a better job than those applying traditional techniques. Our industry colleagues have frequently refuted this claim, arguing that, while highly skilled engineers will perform tasks well given the most elementary of tools and processes, the world is not full of excellent engineers. Most of us benefit from having tools and processes that embody the wisdom gained by previous projects and, dare we say it, more capable colleagues.

To live up to this value statement, it is required that the team members are technically competent in using efficient tools to develop working software for the customer in short

[6] `http://agilemanifesto.org/principles.html`

[7] The aphorism *"A fool with a tool is still just a fool"* is sometimes attributed to Grady Booch.

periods of time. We wonder if all our fellow formalists appreciate the levels of competence among software engineers and hence the work required to deliver methods that can be used in the majority of products. Numerous research projects have demonstrated the potential of new formal methods and tools, such as Rodin[8]. However, the process of bringing them to a state where they are deployable even by R&D engineers requires a major effort, as seen in Deploy[9].

The second principle above refers to "soft" skills and particularly to collaboration and communication. Given engineers who have a willingness and ability to communicate easily among themselves, the tools supporting formal modelling have to make it easy to share models and verification information where appropriate. But how many formal methods tools integrate well with existing development environments, allow models to be updated and easily transmitted, or even exchanged between tools? We feel that collaborative modelling and analysis is not given enough attention in formal methods research.

Most formal methods tools originated in academic research and few have been matured for industrial use. As a result, the focus has been on the functionality afforded by the tools at the expense of the accessibility or user friendliness. If formal methods are to move a step closer to agility, the tool support needs to become easier to pick-up and start using, so attention can be put back on the people actually doing the formal models instead of the tools' limitations. This argues for increased automation and research effort being put into the interaction between modelling and verification tools and the human.

4.2 Working Software over Comprehensive Documentation

The second value statement of the agile manifesto asserts that, whilst good documentation is a valuable guide to the purpose and structure of a software system, it will never be as important as running software that meets its requirements. The value statement is supported by the following principles:

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Working software is the primary measure of progress.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

We suggest that adherence to these principles is probably easier in a project that is kept in-house, rather than a major distributed software development with extensive subcontracting. For large projects with many person-years of work involved, documentation is indispensable and is often a crucial part of the contract between the developer and customer.

[8] `rodin.cs.ncl.ac.uk`
[9] `www.deploy-project.eu`

For the formal methods practitioner, these must be some of the more difficult agile principles to accept. Much of the work on model-oriented formal methods emphasises the quality and clarity of models and the production of proven refinement steps on the way to well engineered code, where proofs document the correctness of the design steps. An agile process adhering to the principle above is more likely to be based on rapid development cycles in which the quality of the formal models produced takes second place to the rapid generation of code. In turn, this may mean that models will tend to be more concrete and implementation-specific than necessary. There is a risk that the development of system models by accretion on each development iteration will end up with models, and of course proofs, that are much too hard to understand or verify to serve a useful purpose.

Although we have reservations about the wisdom of combining formal modelling with rapid iterative code development, our experience would not suggest it should be written off as impossible. However, current formal modelling technology is not geared to such rapid processes. Indeed our own iterative methodology for developing VDM models of real-time systems defers code production to a late stage (although the models themselves can be executable) [LFW09]. Methods like ours should certainly be updated if support for a more iterative approach is desired.

Automation is once again a key factor: where code can be automatically generated, the model may become the product of interest. Further, the benefits of the model must be seen to justify its production costs, for example by allowing automatic generation of test cases, test oracles or run-time assertions. An agile process that wants to gain the benefits of formal modelling techniques has to be disciplined if the formal model is to remain synchronised to the software produced. It is worth noting that nothing in this approach precludes the use of formal methods of code analysis, for example to assist in identifying potential memory management faults. Here again, the high degree of automation can make it an attractive technology.

In general one can say that this value statement is most applicable with executable models where one then needs to be careful about implementation bias [HJ89, Fuc92, AELL92]. From a purist's perspective this is not a recommended approach. As a consequence, formal refinements from non-executable models cannot be considered agile since potentially many layers of models may be necessary before one would be able to present anything to the customers that they can understand [Mor90, BW98, Abr09].

4.3 Customer Collaboration over Contract Negotiation

The third value statement of the agile manifesto, while recognising the value of a contract as a statement of rights and responsibilities, argues that successful developers work closely with customers in a process of mutual education. Two of the agile principles would appear to relate to this value statement:

- Business people and developers must work together daily throughout the project.
- Agile processes promote sustainable development. The sponsors, developers, and

users should be able to maintain a constant pace indefinitely.

This value statement has been criticised on the grounds that large projects will not be initiated before contracts are drawn up and prices agreed. Changes involve renegotiation, and this is not done lightly.

Close customer collaboration has been a feature of several successful industrial formal methods projects [LFB96, vdBVW99, SL99, KN09]. However, in these projects formal methods have only been used as a high-level executable model of the system and no advanced formal methods techniques such as verification have been applied. However, in order to successfully exploit collaboration between business people and formal methods specialists interpersonal skills for this kind of multi-disciplinary teamwork is essential.

A major weakness of many formal methods tools is the inability to attach a quick prototype GUI to a model giving domain experts the opportunity to interact directly with the model and undertake early validation without requiring familiarity with the modelling notation. Actually, this general idea was implemented over 20 years ago in the EPROS prototyping environment [HI88]. Recent extensions to Overture [LLR+10] allow the designer to create a Java applet or JFrame that can act as a graphical interface of the VDM model. Many formal methods modelling tools could benefit from similar extensions if they aim to support some of these agility principles.

4.4 Responding to Change over Following a Plan

The fourth value statement of the agile manifesto acknowledges that change is a reality of software development. Project plans should be flexible enough to assimilate changes in requirements as well as in the technical and business environment. The following two agile principles are relevant here:

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Formal methods have no inherent difficulty in coping with requirements change. Indeed formal models may form a better basis for predicting the consequences of radical changes than attempting an analysis of complex code. However, when formal models are adjusted, the associated validation and verification results will need to be redone if full value is to be gained from the model. Thus, the speed of repeat analysis, and the extent of automated tool support are paramount.

As can be seen, applying the agile principles directly is not something that fits every type of project, but this does not mean that some agile practices cannot be applied to large projects. On the contrary, our experience is that agile development is most often used as an element in a hybrid configuration. For example, initial requirements might be analysed

in a model-based, documented way, while day-to-day development might employ an agile method like SCRUM [Sch04].

We are not convinced that formal methods and formalists "embrace change". Black et al. state [BBB+09] that "formal methods cannot guarantee against requirements-creep". While the concept of requirements creep has negative connotations associated with confused and drifting functionality, being ready to address changing requirements is a necessary part of the agile mindset. Coping adequately with requirements change in a formal setting requires models that are well structured and mechanisms for checking validity that are rapid. Further, to be able to respond to changes in a quick and seamless manner, formal methods practitioners need to accept that models can be less than perfect, especially in the early stages of an iterative development.

4.5 Two remaining principles of technical excellence

Two of the 12 principles do not fit the value statements quite so easily as the others listed above. Both of them deal with aspects of the technical quality of the product:

- Continuous attention to technical excellence and good design enhances agility.
- The best architectures, requirements, and designs emerge from self-organizing teams.

Formal techniques certainly support this focus on quality. Black et al. [BBB+09] also mention the potential synergy between agile and formal methods, opening up the possibility of agile methods being applied to more safety critical domains – something which is currently, to a large extent, off limits to pure agile methods. We conjecture that the second value statement of the original agile manifesto (working software over documentation) has provided a pretext for hack-fixing and ad-hoc programming to call itself an agile process. This has hurt the reputation of agile development, and we would suggest that the addition of a fifth principle favouring quality of the end product over ad-hoc solutions could prevent some of the abuses of agility.

5 Concluding Remarks

The improved analytic power of formal methods tools and greater understanding of the role of rigorous modelling in development processes are gradually improving software practice. However, the claim that formal methods can be part of agile processes should not be made lightly. In this paper, we have examined the value statements and supporting principles of the agile manifesto and have identified areas in which formal methods and tools are hard-pressed to live up to the claim that they can work with agile techniques. In doing so, we have drawn on our own experience of developing and deploying a tool-supported formal method in industrial settings.

Formal methods should not be thought of as development *processes*, but are better seen

as collections of techniques that can be deployed as appropriate. For the agile developer, it is not possible to get benefits from formalism unless the formal notation or technique is focused and easy to learn and apply. Luckily, formal modelling and analysis does not *have* to be burdensome. For example, forms of static analysis and automatic verification can be used to ensure that key properties are preserved from one iteration to the next. For this to be efficient, and to fit into the agile mindset, analysis must have automated tool support. Formalists should stop worrying about development processes if they want to support agility. Instead, they should start adjusting their "only perfect is good enough" mindset, and try a more lightweight approach to formal modelling if their goal is to become more agile.

Formalists may need to remember that most engineers, even good ones, have no prior experience of formal methods technology and demand tools that give answers to analyses in seconds. Developers of formal methods must give serious attention to their ease of use if they are to claim any link with agile software development.

Tools must integrate almost seamlessly with existing development environments if they are to support agile processes and there is considerable research required to make this a reality. Progress can certainly be made by improving tools, in particular in combining with GUI building tools and for automation of different forms of analysis. In fact we would like the agile thinking to go beyond the software to encompass collaboration between different engineering disciplines involved in a complex product development, as in the embedded systems domain [BLV+10].

The agile manifesto is not necessarily consistent with a view of formal methods as correct-by-construction development processes. However, there are good reasons for combining agile principles with the formal techniques. Formal methods researchers and tool builders must, however, address some deficiencies if the benefits of such a collaborative approach are to be realised.

Acknowledgements We are grateful to the organisers of the 2010 edition of the workshop on formal methods and agile methods FM+AM'10 for the invitation to give an invited talk, and to our colleagues in the EU FP7 project DESTECS. Fitzgerald's work is also supported by the EU FP7 Integrated Project DEPLOY and by the UK EPSRC platform grant on Trustworthy Ambient Systems. Wolff's work is supported by the Danish Agency for Science Technology and Innovation. Finally we would like to thank Nick Battle for proof reading a draft of this paper.

References

[Abr09] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To appear. See also http://www.event-b.org.

[AELL92] Michael Andersen, René Elmstrøm, Poul Bøgh Lassen, and Peter Gorm Larsen. Making Specifications Executable – Using IPTES Meta-IV. *Microprocessing and Microprogramming*, 35(1-5):521–528, September 1992.

[ASRW02] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods Review and analysis. Technical Report 478, VTT Technical Research Centre of Finland, 2002.

[BBB+09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest? *IEEE Computer*, 42(9):37–45, September 2009.

[BFL+94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[BLV+10] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design Support and Tooling for Dependable Embedded Control Software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*. ACM, April 2010.

[BW98] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[CJM91] Peter Lindsay Cliff Jones, Kevin Jones and Richard Moore, editors. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[dBSK10] Vieri del Bianco, Dragan Stosic, and Joe Kiniry. Agile Formality: A "Mole" of Software Engineering Practices. In Stefan Gruner and Bernhard Rumpe, editors, *Proc. AM+FM'2010*, Lecture Notes in Informatics. Gesellschaft für Informatik, 2010. To appear.

[DCN+00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. Springer-Verlag, Lecture Notes in Computer Science volume 1785.

[EC03] George Eleftherakis and Anthony J. Cowling. An Agile Formal Development Methodology. In *Proc. 1st. South-East European Workshop on Formal Methods, SEEFM'03*, pages 36–47. Springer-Verlag, 2003.

[FL07] J. S. Fitzgerald and P. G. Larsen. Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007)*, 2007. Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[FLS08] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[FLV08] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[Fuc92] Norbert E. Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, pages 323–334, September 1992.

[HI88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley, 1988.

[HJ89] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[JW96] Daniel Jackson and Jeanette Wing. Lightweight Formal Methods. *IEEE Computer*, 29(4):22–23, April 1996.

[KCN08] Taro Kurita, Miki Chiba, and Yasumasa Nakatsugawa. Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods*, Lecture Notes in Computer Science, pages 425–429. Springer-Verlag, May 2008.

[KN09] T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.

[LDV97] Peter Gorm Larsen Lionel Devauchelle and Henrik Voss. PICGAL: Lessons Learnt from a Practical Use of Formal Specification to Develop a High Reliability Software. In *DASIA'97*. ESA, May 1997.

[LFB96] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[LFW09] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Systems using VDM. *International Journal of Software and Informatics*, 3(2-3), October 2009.

[Liu09] Shaoying Liu. An approach to applying SOFL for agile process and its application in developing a test support tool. *Innovations in Systems and Software Engineering*, 6:137–143, December 2009.

[LLR+10] Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.

[Mar06] Martín López-Nores and José J. Pazos-Arias and Jorge García-Duque and others. Bringing the Agile Philosophy to Formal Specification Settings. *International Journal of Software Engineering and Knowledge Engineering*, 16(6):951–986, 2006.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[NE05] Nan Niu and Steve Easterbrook. On the Use of Model Checking in Verification of Evolving Agile Software Frameworks: An Exploratory Case Study. In *3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2005)*, pages 115–117, Miami, Florida, USA, May 2005. INSTICC Press.

[OMP04] Jonathan S. Ostroff, David Makalsky, and Richard F. Paige. Agile Specification-Driven Development. In J. Eckstein and H. Baumeister, editors, *XP 2004*, volume 3092 of *Lecture Notes in Computer Science*, pages 104–112. Springer, 2004.

[PC86] D.L. Parnas and P.C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, 12(2), February 1986.

[PT99] Armand Puccetti and Jean Yves Tixadou. Application of VDM-SL to the Development of the SPOT4 Programming Messages Generator. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 127–137, September 1999.

[Sai96] Hossein Saiedian. An Invitation to Formal Methods. *IEEE Computer*, 29(4):16–30, April 1996. Roundtable with contributions from experts.

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*. Prentice Hall, 2004. ISBN: 073561993X.

[SL99] Paul R. Smith and Peter Gorm Larsen. Applications of VDM in Banknote Processing. In John S. Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice: Proc. First VDM Workshop 1999*, September 1999. Available at www.vdmportal.org.

[SL10] Fritz Solms and Dawid Loubser. URDAD as a semi-formal approach to analysis and design. *Innovations in Systems and Software Engineering*, 6:155–162, 2010.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An Incremental Methodology for Developing Formal Models. *ACM Transactions on Design Automation of Electronic Systems*, 10(4):589–609, October 2005.

[TFR02] Dan Turk, Robert France, and Bernhard Rumpe. Limitations of Agile Software Processes. In *Proceedings of the 3rd international Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, pages 43–46, May 2002.

[vdBVW99] Manuel van den Berg, Marcel Verhoef, and Mark Wigmans. Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 85–93, September 1999.

[VHL10] Sander Vermolen, Jozef Hooman, and Peter Gorm Larsen. Automating Consistency Proofs of VDM++ Models using HOL. In *Proceedings of the 25th Symposium On Applied Computing (SAC 2010)*, Sierre, Switzerland, March 2010. ACM.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

Part II

Reviewed Papers

- *Agile Formality – A Mole of Software Engineering Practices*
- *State-based Coverage Analysis and UML-driven Equivalence Checking for C++ State Machines*
- *Improved Underspecification for Model-based Testing in Agile Development*
- *An Experience on Formal Analysis of a High-Level Graphical SOA Design*

Agile Formality: A "*Mole*" of Software Engineering Practices

Vieri del Bianco and Dragan Stosic
University College Dublin, Ireland
vieri.delbianco@ucd.ie and dragan.stosic@gmail.com

Joseph R. Kiniry
IT University of Copenhagen, Denmark
kiniry@itu.dk

Abstract:
Members of the agile programming and formal methods communities do not always see eye-to-eye. These two communities often do not talk to or learn from each other. Only recently, as highlighted by the September 2009 issue of IEEE Software, the IFIP workshop on balancing agility and formalism in software engineering, and the first edition of the international workshop for formal methods and agile methods, ideas from the two communities begun synthesize. While the problem-solving approaches and psychological attitudes of members of the two communities differ widely, we exploit this clash of viewpoints, creating a new development processes that actually blends, rather than mashes together, best practices from the two worlds. This paper summarizes our process and a supporting complex case study, showing that it is not only possible, but tasty, to combine the "chili pepper" of formal methods and the "chocolate" of agile programming, thus producing a tasty "*Mole*" (as in the highly-spiced Mexican sauce) of software engineering practices.

1 Introduction

Agile and formal development methodologies usually do not blend together well. This is because of several reasons, the most important of which is often characterized as a radical difference in psychological attitudes about software development.

Formal methodologists often favour an in-depth, think-first approach, where the problem is understood, formalized, and solved; usually, but not always, adopting a waterfall-style of development. Once a formalization is developed, development and verification proceed in an automated and interactive fashion. Consequently, projects that use formal methodologies (FMs) typically focus on critical systems with fixed requirements and somewhat flexible deadlines ("We will ship it when it is right.").

Agile methodologists favour an highly incremental and iterative approach, specifically tailored to cope with changing requirements and precise deadlines. In projects that use agile methodologies (AMs), it is often the case that the problem is only partially understood, and there is a focus only on the aspects that must be implemented in the current develop-

ment iteration. The solution initially developed is usually not optimal, but as it is refined through continuous code refactorings, its quality and validity improve. Test suites are used for several purposes, the most important are system verification, system documentation, and requirement specification, and they are strictly handmade. AMs are typically used in development settings where changing requirements and rapid delivery of the product are paramount.

The two worlds seem irreconcilable. Nevertheless, problems exist that would greatly benefit from both of them [BBB+09]. The problem class in which we have particular interest have *unstable requirements*, are *constrained by deadlines that cannot be postponed*, but also have *subsystems that must be formally specified and verified*.

Terminology In this paper the IEEE 610 [JM90] standard terminology for validation and verification is used. Paraphrasing the standard, *verification* is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed, according to the specifications ("you build the system in the right way"); *validation* is the process of evaluating software to determine whether it satisfies specified requirements ensuring it meets the user's needs ("you build the right system").

Within the FMs community, verification and validation have different meanings than in the IEEE 610 standard. Loosely *verification* (in the paper *formal verification*) means the use of formal methods to formally, statically verify that a system conforms to its specification; *validation* (in the paper *formal validation*) means the use of traditional software engineering practices to check, statically or dynamically, that a system conforms to its specification (whether in the form of tests, requirements, etc.) by hand or via execution under some execution scenarios. In the present paper both perspectives are respected and acknowledged.

Tests are usually classified by dichotomies: unit vs. functional, black-box vs. white-box, testing vs. design, tester vs. coder. Yet testing does not always fit into these dichotomies, especially in the case of agile testing approaches [Bec07]. The black-box vs. white-box dichotomy is particularly problematic, since box boundaries change one's perspective. In this paper, functional tests are always black-box tests, coverage tests are always white-box tests, generated tests are always black-box tests (the box is the module specified); other strict rules do not hold.

Additionally, "...development of a system," more precisely means "...analysis, design, development, verification, formal validation of a system."

1.1 Case Study

Proposing new practices with no supporting evidence is an uninteresting proposition. A complex case study is mandatory to test-bench the proposed practices. Our case study focuses on the development of a device driver to control and communicate with an embedded custom circuit board equipped with more than a dozen sensors. The communication protocol with the board is asynchronous and packet-based and the board is novel and

custom-designed. The protocol is focused on controlling the board and reporting sensor measurements and board status.

This software driver has been developed using our process while satisfying seemingly contradictory requirements: (1) the protocol implementation must be thoroughly verified; (2) parts of the protocol must be formally specified; (3) a board simulator must be designed, implemented, and verified to match the specifications of, and actual behaviour of, the physical circuit board; (4) if board components change during development, the associated formal specifications and the simulator whose behavior reflects such changes must be updated accordingly; (5) there are strict software delivery deadlines. Consequently, these process requirements constitute an environment that benefits from both agile and formal approaches to software development.

A novel development process has been drafted to cope with the contrasting requirements of the case study, blending agile and formal practices. In the following, the development process and the practices are summarized, and their successful application to the case study is reported, yielding several promising results.

1.2 Formal and Agile Integration

To enable the integration of the two worlds, the highly iterative and incremental approach found in most of the AMs must be maintained, and formal validation methodologies, traditionally used in a waterfall development process (formally specify the system, implement the system, validate the implementation against the specification) must be adapted to a highly iterative one.

The most common software verification practices in AMs must be considered when trying to integrate FMs and AMs since they are a fundamental part of the development process. The most significant verification related practice is Test Driven Development (TDD) [Bec03]. TDD is a software development technique, originally defined in the Extreme Programming (XP) [BA04] methodology and is characterized by a process that focus on writing unit tests of a module before writing the code of said module.

The test driven approach relies on writing tests before implementations. It is applicable to a very small scale and large scale cycles In the former case, TDD yields to activity cycles as short as a few minutes. In the latter, requirements are immediately translated into functional tests resultin in activity cycles as long as a full delivery iteration, which is usually no longer than a couple of months. The test driven approach is a cornerstone of XP, but it is so popular that has been adopted in many other AMs as well. It is also considered a good software development technique when used on its own, regardless of the enclosing development process.

Some tentative attempts to reconcile FMs and AMs have been developed by others. The reoccuring theme of these attempts is that idea that handmade test suites are no longer used, but instead are replaced by different kinds of automated formal validation based on a system level formal specification. Automated validation can be as simple as enabling runtime checking of assertions or automatically generating a test suite based on the sys-

tem properties, or as complex as proving the whole system's behaviour through formal verification.

Eleftherakis and Cowling in [EC03] propose XFun, an iterative and incremental process to generate X-Machines models, an extension of Finite State Machines. The development process impose a complete specification of the system, and its verification is done exclusively using the tools provided by the X-Machines FM. The development process proposed by Herranz and Moreno-Navarro [HM03] is quite similar. They propose an iterative process to model a system using SLAM-SL, an object-oriented specification language supported by tools. Some XP practices are fully adopted or considered compatible, like pair-programming, iterative and incremental development and system refactoring; while tests are completely replaced by the verification tools provided by the SLAM suite. A different FM, but similar approach, is found also in the work of Suhaib [SMSB05] where XFM methodology is introduced.

As already mentioned, in all these attempts, the test suites are automatically generated using system formal specification, or are completely replaced by formal verification, typically through the use of theorem provers and/or static checkers. The major precondition on the use of these approaches is, at the minimum, a specification of all the relevant aspects of the system under development. Such a specification enables simple verification techniques such as runtime checking (via assertions and design by contract [Mey97] using preconditions, postconditions and invariants) and the automatic generation of test suites [CL02a, CL04, CKP05]. Via a complete and sound system specification, supported by an appropriate formal language (to specify the system model and its properties) coupled with a (possibly restricted) programming language, one can statically prove properties about the system using a variety of tools and techniques [CH02, DNS05, KC05].

The underlying idea of this approach is simple: what was once the purpose of handwritten tests (verification, documentation and requirement specification) are now the respnsability of the formal specifications, and a complete formal specification of the system under development is needed to apply these methodologies. The problem is that this requirement is very difficult to fulfill since the effort required to write a complete formal specification of the system in most real world complex cases is usually greater than writing a suite of tests [Gla02]. If a complete formal specification is not available, all of the previous methodologies share a common problem: the parts informally specified are not verified at all. These parts os the system are consequently verified with traditional methods, but the connections between development artifacts and related activities (code, tests, design, refactoring) are neither detailed nor enforced. This is problematic as the loops and relations between the different development artifacts are the inner engine of many AMs, as they impose an iterative and incremental pace to the development process.

Liu takes a different approach wherein the SOFL methodology [Liu04] is merged into agile processes [Liu10]. The SOFL methodology is based on a three step specification approach: (1) informal specification, (2) semi-formal specification, (3) formal specification. The agile adaptation consists mainly in introducing shorter loops between activities and reducing the size of the set of formal specifications. The formal specifications are used only to help understand ambiguous statements in the semi-formal specification and are not maintained while the system evolves. Testing and inspections are based on the

semi-formal specification. The formal specification are partial and are only used for documentation purposes and than discarded—they are not used at all in the verification of the system.

I contrast our objective is to formally specify only parts of the system (in order to cope with constrained resources and unstable requirements) and to develop and refine a highly iterative and incremental development process that blends formal and agile practices.

The main open problem unaddressed in the literature is how to connect development artifacts of the two worlds of AMs and FMs together. To answer this challenge we define activity cycles, similar to what is found in AMs (especially in XP), to solve this problem in an highly iterative development process. We show how tests drive the formal specification, how the formal specifications drive tests and code development, how handmade and automated tests coexist and support each other, how the unspecified parts of the system are incrementally specified. That is, we show how to blend, and not just tack together, formal validation and agile verification practices: a "*Mole*" of verification practices[1].

2 A Real and Complex Case Study: Rapid Development in Small-scale Hardware Software Co-design

"*Mole*" practices are applied in the context of a real world case study that matches the problem requirements previously detailed: unstable requirements and development artifacts that need to be formally specified and validated.

2.1 The UCD CASL SenseTile System

The **UCD CASL SenseTile System** is a large-scale, general-purpose sensor system developed at the University College Dublin in Dublin, Ireland. The facility provides a scalable and extensible infrastructure for performing in-depth investigation into both the specific and general issues in large-scale sensor networking. This system integrates a sensor platform, a datastore, and a compute farm. The sensor platform is a custom-designed but inexpensive sensor platform (called the *SenseTile*) paired with general-purpose small-scale compute nodes, including everything from low-power PDAs to powerful touchscreen-based portable computers. Besides containing over a dozen sensors packaged on the *SenseTile* itself, the board is expandable as well, as new sensors are added to it easily.

The case study is focused in building the sensor board and its software driver concurrently. Because of hard time constraints and the initial unavailability of the custom board, we must progress concurrently with all the development tasks, including: (1) specification of the communication protocol; (2) specification of the physical sensor board; (3) development and fabrication of the physical board; (4) development of the embedded software for the

[1] A mole (*'mōlā*) is a highly spiced Mexican sauce made chiefly from chili peppers (agile) and chocolate (formal), served with meat (working system).

board; (5) development of the communication protocol software driver of the board; and (6) development of software simulators of the board.

The development of the physical board, along with its embedded software (development tasks 2, 3, 4), is carried out by a third party under our guidance, thus it is not directly taken into account here. The specification of the communication protocol (development task 1) is a joint effort between ourselves and the third party, while the remaining tasks (5, 6) are performed in isolation. Dependencies are straightforward: the specification of the communication protocol (1) is the most stable element, but it still depends on the sensor board (2, 3, 4): large changes in the latter affect the former. Additionally, the driver and the simulator depends directly on the communication protocol.

Communication and synchronization with the external manufacturer is frequent but not optimal. The main synchronization artifact is the protocol specification, which remains stable at high-level, but is changed frequently in the low-level details. A common domain language has been found and agreed upon, but the FMs and programming platforms differs widely. Consequently the informal specification of the communication protocol is the most reliable source of information.

Related Methodologies There exist various approaches in literature addressing this kind of development constraints (rapid development in hardware software co-design), but all are focused on large-scale systems.

The hardware-software formal co-design methodologies usually have several common development artifacts [SDMH00, HKM01] including a high-level specification of the system, a translation (refinement) of the high-level specification to low-level ones (hardware and software counterparts), the possibility to generate software code from the low-level specifications, a hardware simulator capable of simulating an hardware component based on hardware specification, the hardware component developed.

When considering the development of the device and of its software driver as separate entities that possibly have to be developed concurrently, the existing approaches are similar to the ones seen in the case of hardware-software co-design [Val95, SM02, RCKH09], all of which focus on a specification that aims to be as complete as possible. A complete specification is neither feasible nor convenient in our case. The protocol specification must be enhanced incrementally and its complete specification cannot be provided. This makes the problem an ideal candidate to test our "*Mole*" of practices.

2.2 The Chosen Formal Methodology: Formal Specifications with JML

An appropriate FM must be chosen. It must be able to support a specification that is built incrementally, consequently formal methods that demand a complete specification must be avoided. It must be able to automatically generate tests. And finally the formal method must be complemented by tools that support the verification of system properties at runtime.

The Java programming language and platform are chosen for the implementation, the Java Modeling Language (JML) [LBR01] is used to write formal specifications, and JML2 [LPC+04] tool suite is used as supporting tool [BCC+05]. The JML2 tool suite includes runtime assertion checker [CL02b] and a unit test generator [CL02a]. A complete specification is not required to use JML2 tool suite effectively.

JML is a rich behavioural interface specification language (BISL) and focuses on the modular specification of classes and objects [LBR99]. JML includes standard specification constructs like assumptions, assertions, axioms, and pre- and postconditions, framing clauses, and invariants. It also includes a rich model-based specification layer that includes mathematical models, data refinements, datagroups, and many other advanced object-oriented specification features [Cha06]. Many tools, ranging from compilers to static checkers to full-functional verification systems support the JML language [BCC+05].

JML is sometimes used in a Design-by-Contract style, where a specification is written from scratch, reusing existing APIs that have specifications of their own, and then an implementation is written conforming to that specification [Mey92]. At other times an existing piece of implemented and tested code is annotated with specifications after-the-fact.

The two tools used most frequently to check the correctness of implementations are the JML Runtime Assertion Checker (RAC) and the Extended Static Checker for Java, version 2 (ESC/Java2) [CL02b, BCC+05, KC05]. The former compiles executable assertions into runtime checks. The latter tool performs automated modular semantic static analysis to attempt to prove that the program under analysis does not misbehave and conforms to its specification (lightweight functional correctness).

3 Blending Formal And Agile Development: the "*Mole*" Practices

AMs are all based on a highly iterative and incremental process. They share a common approach on team management, customer relation, simplification and removal of unnecessary artifacts and activities, they rate working solutions and customers satisfaction as the most important indicators considered during development. The Agile Manifesto [BBvB+01] summarizes the philosophy and principles shared by all AMs.

The Agile Manifesto does not suggest any specific development techniques, it describes AMs from a very high abstraction level. Nevertheless, most AMs share a similar approach to artifact verification. Functional test suites, used as precise requirements or user stories definition, are applied in DSDM [Sta97], XP and Crystal Clear [Coc04]. Unit tests and TDD are mentioned, and often included, in most of the AMs defined so far.

High level test suites, composed of functional tests, describe requirements and bind together requirement documentation and system behaviour: an informal requirement is supported by a set of functional tests. The associated development cycle is one iteration long. Low level test suites, composed of unit tests, describe the module behaviour and bind together code documentation, code and the contract imposed on the module: unit tests represent the contract and the documentation. The associated development cycle is less

than one day (as short as a few minutes).

Continuous refactoring [FBB+99] enables iterative and incremental development, since the system architecture and design, whether explicitly specified or not, must be cleaned up and modified, to accommodate new requirements, services and modules. Refactoring is possible and feasible because of the safety nets provided by the test suites, used as regression test suites.

Our objective is to formally specify only parts of the system under development, in an incremental iterative process. Formal artifacts must coexist with informal ones: requirements, expressed as functional tests or as formal specifications, handmade and generated tests, documentation, source code. The development process needs to guarantee consistency over all the involved artifacts.

Both Test Driven Development (TDD) and classical Formal specification Driven Development (FDD) guarantee consistency on all the involved artifacts. Both TDD and FDD support iterative development. The development cycle adopted in TDD is shown in Figure 1 and the one adopted in classic FDD is shown in Figure 2. Unit tests used in TDD are substituted in FDD by formal specifications supported by the verification environment; in our case the verification environment is the generated unit test suite. Formal specifications replace unit tests: they specify, document and verify the system under development.

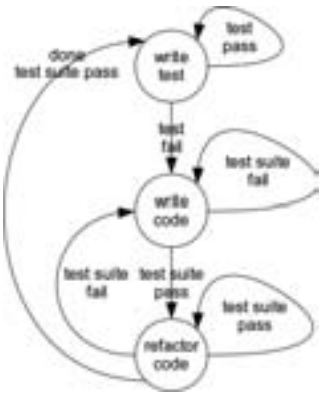


Figure 1: TDD development cycle.

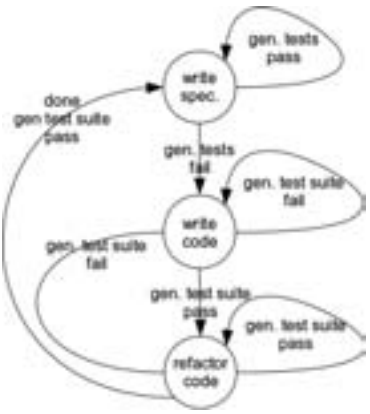


Figure 2: FDD classic development cycle.

FDD is used for the formal specified parts of the system, whether TDD is used for the remaining ones: a "composition" of formal and agile verification practices. Yet there are several problems that still need to be addressed. No communication between the two worlds is permitted, it is not defined how to formally specify a module after it has been implemented and verified in a non formal way. A module only partially specified is a sort of hybrid that must be treated accordingly. Finally, in the FDD cycle shown in Figure 2, the implicit assumption is that it is always possible to identify the correct formal specifications of the system, but this is not always straighforward. Identifying the correct formal specification is a difficult task, it has to be supported and verified by the development process.

A partially specified module cannot be verified properly only through automatically generated tests. Handmade tests have to be used together with generated ones. Handmade tests verify complex behaviours that are informally specified, or complex behaviours that are formally specified but cannot be replicated by the formal verification tools, because of the limits of the verification tools themselves. The constraints imposed by formal verification tools [Gla02] must be taken into account when deciding whether supporting the code with handmade tests, and verification tools based on automatically generated tests are not an exception. The development cycle able to blend formal specifications, handmade tests and code development is shown in Figure 3: the formal specification drives the handmade tests. Both handmade tests and the formal specification drive the code development.

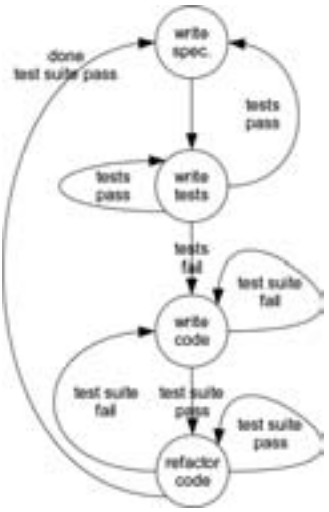


Figure 3: FDD development cycle with handmade tests.

Figure 4: TDF development cycle.

The opposite path still needs to be covered, moving incrementally from a tested code to a formally specified code. In this case, new handmade tests and existing working code drives the formal specification, this is what we call Test Driven Formal specification (TDF). The resulting development cycle code is shown in Figure 4, TDF is used to incrementally add formal specifications to an existing working system.

When the "*Mole*" verification practices are used together the initial objectives are achieved: an incremental and iterative process, where development pace is defined by very short development cycles similar to TDD; the complete freedom to decide what is formally specified; the development cycles and the associated verification practices maintain the corresponding artifacts involved consistent, the correct development cycle is determined by the formality of the artifacts involved, ranging from pure development cycles, shown in Figure 1 and 2, to blended ones, shown in Figure 3 and 4.

4 "*Mole*" Verification Practices Applied

4.1 Data Stream, Protocol Description and Protocol Specification

The *Sensor Board* protocol is asynchronous and packet-based; the packets have a fixed length. The protocol is separated in multiple layers to ease its implementation in the driver. The layers identified are the following: (1) packet byte structure: the internal representation of the packet; (2) packet info structure: the meaningful fields contained in the packets; (3) single packet rules: the content acceptable values, and how they influence each other; (4) packet sequence rules: the content acceptable values, based on values of previously received packets; (5) packet input output asynchronous rules: the content acceptable values and reaction constraints on output packets, based on previously transmitted input packets. The first three layers are analyzed in the case study, considering only the output sensor data packet.

The packet byte and info structure of the output sensor data packet reflect the board capabilities and built in sensors. A single packet has to accommodate various types of data: fast, medium and slow data rate streams, together with metadata describing the sensors and the board state. The internal structure of the packet is strictly fixed.

A packet is internally composed by 82 frames. A frame accommodates data from the fast and medium data rate streams (4 fast data rate stream and 8 medium data rate stream channels) and theirs associated metadata.

The single packet rules delimit the boundaries of the values obtainable from the packet: each defined sensor represented in the packet, as well as the metadata describing the *SenseTile Sensor Board* and the packet and frame contents, are constrained by a range of acceptable values. There are also rules affecting more than one value, and rules specifying a correct sequence of frames.

The specifications of the protocol are distilled and refined incrementally. The protocol is divided in various (thin) layers, and each of the layer is verified with a different approach: some of the layers are specified formally.

Packet byte structure specification Verification is obtained through an handmade test suite, composed of unit and integration tests; the tests specifies the behaviours of the implementation, which is capable of recognizing the proper packet structure in a binary data stream. The packet byte structure is not stable: the internal byte structure changed several times during development. Therefore, packet byte structure is informally specified.

The test suites are used to verify board simulators (low-level), but they cannot be used to check properties at runtime. Code and tests are implemented using TDD.

Packet info structure specification Verification is provided by handmade test suite, JML annotations and generated unit test suite. It has been built starting from handmade tests only (TDD development cycles). Later on JML annotations have been added incrementally, using a mix of "*Mole*" practices: to move from handmade tests to formal

specifications, TDF have been initially used; than all the "*Mole*" practices have been used according to the artifacts involved.

The test suites are used to verify board simulators (high-level), RAC and JML annotations are used to check the formally specified properties at runtime. Code and tests have been initially implemented using TDD, than using all the "*Mole*" verification practices.

Single packet rules The verification is provided exclusively by the formal specification with JML language, combined with the generated unit test suite and handmade test suite to verify the most complex behaviour. It is built starting from JML annotations, supported by handmade tests when needed.

The test suites are used to verify board simulators (high-level), RAC and JML annotations are used to check most of the rules at runtime. Code and tests are implemented using both forms of FDD.

On simulators Simulators are used to test parts of the system minimizing dependencies: a simulator can be used by upper layers, with no need to provide the functionalities of the lower layers.

The layer structure is not matched by the corresponding simulators. The driver API is splitted in two abstraction layers: the general high-level interface, that exposes the main functionalities of the board and the main contents of the packets, and the lower level implementation that parses the data streams in and out the board, meant to translate the higher level instructions and data in properly formed packets. The simulators are built according to these abstraction layers. The high-level simulator implements the interfaces providing the methods to deal with an abstracted *Sensor Board*. The low-level simulator is capable to rebuild the *Sensor Board* data streams: the in and out data streams are built exactly as the sensor board is expected to parse or generate.

4.2 JML Specification Examples

The JML specifications are of varying complexities. Some of them are rather simple, focusing on constraints that should hold when calling a method (the preconditions) and constraints on the return value (the very basic form of postconditions). In listing 1 a simple JML specification example is shown, the method `getTemperature` is declared `/*@ pure @*/`, which means that it cannot change the state of any object (a postcondition); the specification also constraints the return value with a lower and upper bound (another postcondition).

The complex specifications usually focus on properties regarding the behaviour of a whole object. In listing 2 a complex invariant example is shown; an invariant is a property maintained during the life cycle of an object, more precisely, an invariant is assumed on entry and guaranteed on exit of each method of an object. The invariant is constraining the number of samples for each medium data rate streams: the total number of streams

Listing 1: A simple specification with JML annotation: simple postconditions.

```
/*@
    ensures \result >= -880;
    ensures \result <= 2047;
@*/
/*@ pure @*/ short getTemperature();
```

is `Frame.ADC_CHANNELS`, the total number of frames is `FRAMES`, the constant constraining the number of samples is `FRAMES/Frame.ADC_CHANNELS+1`, meaning that the samples contained in a frame are fairly distributed on the channels. The valid samples are counted parsing all the frames contained in a packet, selecting only the matching valid samples. A medium data rate stream sample is considered valid when `isADCActive()` method returns **`true`**.

Listing 2: A complex specification with JML annotation: invariant constraining the number of samples for medium data rate streams.

```
/*@
    invariant (
      \forall int channel;
          0 <= channel &&
          channel < Frame.ADC_CHANNELS; (
        \num_of int i;
            0 <= i &&
            i < (FRAMES-1); (
          (getFrame(i).isADCActive()) &&
          (getFrame(i).getADCChannel() == channel)
        )
      ) <= (FRAMES / Frame.ADC_CHANNELS + 1)
    );
@*/
```

4.3 Test Cases

The unit test cases that verify the protocol driver are of two kinds: handmade unit tests and automatically generated unit tests based on JML specifications; they are complementary and built to be used together.

The test effectiveness is evaluated for each test suite; the evaluation is carried out in section 5. The test effectiveness evaluation considers three elements: effective results on piloting the real board (quantitative), code coverage (quantitative), development help and usefulness (qualitative).

Handmade tests The package structure of the driver is shown in Figure 5: two independent packages are defined (`Stream` and `Driver`). The packages are abstract, that is, they mainly contain abstractions; in Java language this is translated into a package which contains mainly abstract classes or interfaces. The dependency between the two abstract packages is not direct, it is realized through an implementation (`StreamDriver`). This is needed to maintain a high decoupling of the packages, and is the result of applying the dependency inversion principle [Mar96].

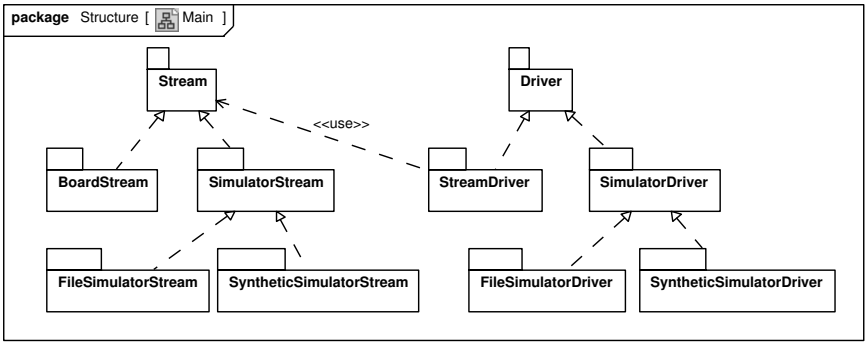


Figure 5: Main packages class diagram.

The test package structure, shown in Figure 6, reflects and mimics the code structure; the tests for an abstract package are abstract, and implemented by the package that tests a corresponding system implementation. This is a well known test pattern (the Abstract Test Pattern [Tho04]) used to test that the contracts defined in the abstractions are respected in all the implementations. For instance, package `DriverT` contains abstract tests for the abstractions of package `Driver`, package `StreamDriverT` inherits the abstractions of `DriverT` and makes them concrete, to test the corresponding implementation `StreamDriver`; package `StreamDriverT` also contains stand alone tests written specifically for the implementation `StreamDriver`.

Generated tests The JML specifications are used to generate tests. The resulting test package structure is closely related and reflects the overall package structure; each package have a corresponding generated test package.

On simulators Simulators are adopted in the handmade test suits as stubs. The resulting structure is shown in Figure 7. For instance, in Figure 7 the test suite `StreamDriverT` is using the `SimulatorStream` to properly simulate the `Stream` parsed by `StreamDriver`, which is the system under test.

The simulators are adopted in both the handmade and the generated test suite; both the suites need a proper set of stubs in order to be executed.

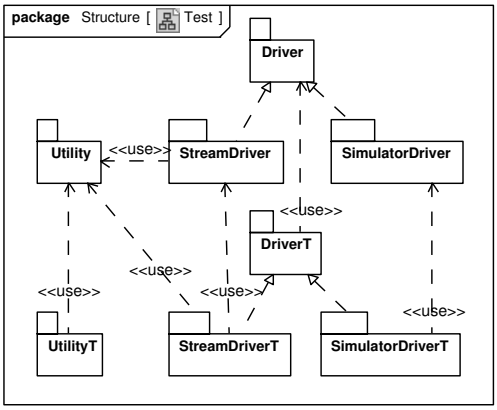


Figure 6: Test packages class diagram.

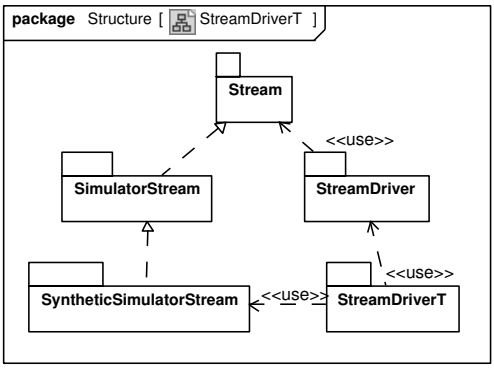


Figure 7: StreamDriver test packages class diagram: use of the stream simulator.

4.4 Test Cases on the Job

The handmade test suite is composed by over 140 tests, while the generated one is 100 times bigger, totalling over 14000 tests. The large number of generated tests are explained looking closely at how the generated tests are created by the JML framework. The generated tests explore the possible input combinations on public methods, combining the type data ranges that are specified for the test suite (see [CL02c] for more details on how the JML framework works on generating unit tests). For instance, let's suppose a method `m(int p1, int p2)` must be tested in a class `C`; the data ranges specified for type **`int`** are `{1,2,3}` and for type `C` are the instances `{c1,c2,c3,c4}`. The JML framework generates $24 = 3 \times 3 \times 4$ tests, since they are the combination of the data ranges involved (the data range of the parameters and the data range of the type owning the method under test). Nevertheless, not all the generated tests are executed: a generated test is not

executed, unless the preconditions of the called method are satisfied. Thus, depending on the preconditions specified for a method, the number of the generated tests that are actually executed in the test suite are significantly smaller compared to the total number of generated tests.

Test suites execution time are different: the handmade test suite, with the runtime assertion checking active, runs completely in less than one minute on an average PC, while the generated test suite runs in more than 20 minutes (the rough ratio is $1 : 60$). Most of the time is spent on setting up the suite; only considering the real execution of the tests, with no suite setup, the execution time reduces to less than 4 minutes (the rough ratio is $1 : 10$).

Code coverage metrics (white-box testing) are used to compare in a quantitative way the effectiveness of the test suites. This is only an indicator, since it is accepted that code coverage alone cannot asses the quality of a test suite[Mar99, CKV06]. Various coverage metrics exist, the test suites are analyzed with statement code coverage (one of the simplest types)[2]: statement coverage reports whether each statement is encountered during execution. The coverage results are reported in the Table 1.

Table 1: Statement code coverage result, categorized by source package.

	hand	***generated***	***total***
Driver	15.9%	6.3%	15.9%
StreamDriver	76%	79.7%	88.5%
SimulatorDriver	64.9%	44.2%	71.2%
Utility	98.1%	74.5%	98.1%

The coverage measures reported in Table 1 do not reach 100%. This is because of the effects of both non public utility methods, and abstract methods in interfaces. The effect of non public methods (package Java visibility) is that these methods are taken into account as public and protected ones are, during code coverage calculation; these methods are not part of the interface, the system does not depend on them, they are only used in object initialization or in object setups performed during unit tests. The effect is mainly seen in low `SimulatorDriver` code coverage figures: the `SimulatorDriver` package has many utility non public methods.

Regarding interfaces, an interface contains no statements, so when an interface method is called, it is the method of the implementation class used that is actually covered. This effect is visible in `Driver` coverage figures: `Driver` package contains interfaces (that are not counted at all) and some very simple real class (exceptions) that are not thoroughly tested.

The statement code coverage figures show that the test suites do a good job on the most important package, the `StreamDriver`. The generated test suite reaches almost 80%, while the handmade tests have only a slightly lower coverage ratio: 76%. The combination of test suites raise the coverage ratio to 88.5%. This is a clear indication that one test suite

[2]The tool used to obtain statement code coverage metrics is Emma.

is not completely overlapping the other: the intuition that test suites are not alternatives, but have to be used together to achieve the higher benefits, is confirmed. A similar indication is provided by `SimulatorDriver` package as well.

The qualitative difference of the two test suites are understood observing how the suites are built. An handmade unit test usually requires objects initialization, one or more method calls to the modules under test, and the following assertions to verify the expected state change on the modules involved. A generated unit test has a more focused and limited scope: only one method call is performed in each test, therefore it is difficult to explore complex behaviours. The test suites give their best when used in a combined approach: the generated test suite, checking simpler and formalized behaviours, and the handmade, checking the more complex ones, whether formalized or not.

Effectiveness on board delivery The first *SenseTile Sensor Board* prototype builds packets with no data, but the packets are built in a way not consistent with the specification. The board errors are detected by using the driver implementation, the `StreamDriver` package.

The second prototype builds packets with values taken from real sensors installed on the board. The only sensors that are not working are the sensors devoted to the fast rate data streams (audio sensors). One error needed to be corrected in the driver implementation, because of a (rare) combinations of conditions not initially covered by the test cases, but that occasionally showed up during real use.

The second prototype respected the packet byte structure specification, but was not fully compliant to the packet info structure specification and the single packet rules. It also randomly generated completely invalid packets when overheated. The formal JML specifications and the runtime assertion checker correctly identified these errors. 4 protocol errors regarding the packet info structure specification, and 2 protocol errors regarding the single packet rules were found, as well as the invalid packets.

5 Retrospective on "*Mole*" Practices Effectiveness

We believe that "*Mole*" practices are relatively easy to introduce and use. Many kinds of projects could benefit from these practices, not only those that have unstable requirements but also demands for a certain level of formality. Even projects with no requirements for formal specification at all benefit from these practices, since some system properties are easy to specify and, through their introduction, there is a corresponding reduction in the test suite development effort.

The cornerstone of the practices is the formal methodology and its supporting tools, and the precondition on the tools' utility is not always easily fulfilled. Methodologies that require a complete formalization are nearly impossibile to apply. In addition, it is not clear if a tool suite that does not provide a runtime verification, either via runtime checking or simulation, can be broadly used succesfully, nor it is clear whether formal methods that are

not based on the contract paradigm are useful in this context. These are open questions that will need to be addressed in the future to determine the broad applicability of the "*Mole*" practices.

There are no strict rules to determine which of the "*Mole*" practices to use when facing a specific problem. Since experience trumps cookbooks, we can only give some advice. We suggest using TDD if there is no need for any specifications at all, FDD with generated tests if the specification are not too difficult and execution scenarios are simple enough to be covered by generated tests, FDD with hand made tests if the scenarios are complex, and TDF when the specification is complex. We feel that the experience we have gathered is not yet enogh to distill richer idioms or patterns—such is future work.

Upon reflect, we find that the end result obtained via combining FDD with hand made tests is remarkable, as it is as natural to work with as TDD. FDD with generated tests is sometimes annoying because of the time needed to run the full test suite, thus often only part of the test suite is executed so that the development loops do not take too long. TDF is great for complex preconditions, but really difficult with postconditions, especially if the specification is written for existing code. When writing after code, the corresponding test is not expected to fail, hence the feedback is limited and the contract is often not as strong as needed. In this specific case, TDF effectiveness is limited.

6 Conclusions

In this paper we focus on a specific set of development challenges with which neither AMs nor FMs are completely confortable. The challenges are characterized by unstable requirements combined with artifacts that must be formally specified and verified all while the developmemnt team is constrained by deadlines that cannot be postponed. We propose a blend of agile and formal engineering practices, enclosed by an iterative and incremental development process: the "*Mole*" development process.

The verification practices described in this paper recreate the fast feedback development environment we find in Test Driven Development in the presence of both formal and informal artifacts. A total of four practices are presented—two of them are conservative, following closely the development practices of FMs and AMs, while the other pair are innovative, blending together elements from both worlds.

The practices have been applied succesfully to develop a driver for a custom embedded sensor board equipped with board simulators and protocol verifiers. And while the protocol and the board specifications were informal and unstable, a working software product was needed as soon as possible, with incrementally added functionality, and ensuring that the software and hardware products were linked via the last specifications of the protocol and the board. "*Mole*" verification practices enabled us to keep in sync formal specifications (JML annotations), informal specifications (test suites) and source code through very fast and short development cycles. We developed handmade test suites and JML annotations, and generated test suites, all of wich succesfully supported the verification of the software driver and the simulators as well as the hardware board when it was delivered.

In fact, the resulting system had no flaws, as it worked correctly the very first time it was plugged in to the prototype hardware.

The practices we describe are general, and are applicable to different development languages and FMs. One important constraint is on the FM, since it must cope with partial specifications. Another constraint is on the tools available, which must verify the system properties at runtime and partially automate the verification process of the formalized properties.

References

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, second edition, 2004.

[BBB+09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, 2009.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. `http://agilemanifesto.org/`, 2001.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, February 2005.

[Bec03] Kent Beck. *Test-Driven Development: By example*. Addison-Wesley, 2003.

[Bec07] Kent Beck. Test-Driven Development Violates the Dichotomies of Testing. `http://www.threeriversinstitute.org/Testing%20Dichotomies%20and%20TDD.htm`, June 2007.

[CH02] N. Cataño and M. Huisman. Formal Specification of Gemplus' Electronic Purse Case Study using ESC/Java. In *Proceedings of Formal Methods Europe (FME) 2002*, number 2391 in Lecture Notes in Computer Science, pages 272–289. Springer-Verlag, 2002.

[Cha06] Patrice Chalin. Early Detection of JML Specification Errors using ESC/Java2. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*. ACM Press, November 2006.

[CKP05] Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perum. A Complete Automation of Unit Testing for Java Programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290—295, 2005.

[CKV06] Hana Chockler, Orna Kupferman, and Moshe Vardi. Coverage Metrics for Formal Verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):373–386, 2006.

[CL02a] Yoonsik Cheon and Gary Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the 16th European Conference in Object-Oriented Programming (ECOOP)*, pages 1789–1901. Springer, 2002.

[CL02b] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[CL02c] Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In Boris Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, June 2002.

[CL04] Yoonsik Cheon and Gary T Leavens. The JML and JUnit Way of Unit Testing and Its Implementation. Technical Report 04-02, Department of Computer Science, Iowa State University, February 2004.

[Coc04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.

[EC03] G. Eleftherakis and A. J Cowling. An Agile Formal Development Methodology. In *Proceedings of the 1st South Eastern European Workshop on Formal Methods (SEEFM)*, page 36â47, 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.

[Gla02] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, October 2002.

[HKM01] A. Hoffman, T. Kogel, and H. Meyr. A Framework for Fast Hardware-Software Cosimulation. In *Proceedings of the European Conference on Design, Automation and Test*, pages 760–765, Munich, Germany, 2001. IEEE Press.

[HM03] Angel Herranz and Juan Moreno-Navarro. Formal Extreme (and Extremely Formal) Programming. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP)*, page 1012. Springer, 2003.

[JM90] F. Jay and R. Mayer. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std.*, 610:1990, 1990.

[KC05] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2005.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.

[LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001.

[Liu04] S. Liu. *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer, 2004.

[Liu10] Shaoying Liu. An Approach to Applying SOFL for Agile Process and Its Application in Developing a Test Support Tool. *Innovations in Systems and Software Engineering*, 6(1):137–143, March 2010.

[LPC+04] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*. Department of Computer Science, Iowa State University, 226 Atanasoff Hall, draft revision 1.94 edition, 2004.

[Mar96] Robert Cecil Martin. The Dependency Inversion Principle. *C++ Report*, 8(6):61–66, 1996.

[Mar99] Brian Marick. How to Misuse Code Coverage. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 16—18, 1999.

[Mey92] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, October 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer systems*, pages 275–288, Nuremberg, Germany, 2009. ACM.

[SDMH00] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann. Hardware/Software Codesign and Rapid Prototyping of Embedded Systems. *IEEE Design & Test of Computers*, 17(2):28–38, 2000.

[SM02] R. Siegmund and D. Muller. Automatic Synthesis of Communication Controller Hardware from Protocol Specifications. *IEEE Design & Test of Computers*, 19(4):84–95, 2002.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An Incremental Methodology for Developing Formal Models. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):589–609, 2005.

[Sta97] J. Stapleton. *Dynamic Systems Development Method*. Addison-Wesley, 1997.

[Tho04] J. Thomas. *Java Testing Patterns*. John Wiley & Sons, 2004.

[Val95] A. Changuel C. A. Valderrama. A Unified Model for Cosimulation and Cosynthesis of Mixed Hardware/Software Systems. `http://www2.computer.org/portal/web/csdl/doi/10.1109/EDTC.1995.470395`, March 1995.

State-based Coverage Analysis and UML-driven Equivalence Checking for C++ State Machines

Patrick Heckeler, Jörg Behrend,
Thomas Kropf, Jürgen Ruf, Wolfgang Rosenstiel
University of Tübingen
{heckeler, behrend, kropf, ruf, rosenstiel}@informatik.uni-tuebingen.de

Roland Weiss
Industrial Software Systems
ABB Corporate Research
roland.weiss@de.abb.com

Abstract: This paper presents a methodology using an instrumentation-based behavioral checker to detect behavioral deviations of a C++ object implementing a finite state machine (FSM) and the corresponding specification defined as a UML state chart. The approach is able to link the source code with the appropriate states and provides a coverage analysis to show which states have been covered by unit, system and integration tests. Furthermore, the approach provides statistical information about the distribution of covered lines of code among all included files and directories. As a proof of concept the presented approach has been implemented in terms of a C++-library and has been successfully applied to OPC UA, an industrial automation infrastructure software.

1 Introduction

In the area of safety-critical hardware-depended software (SCHDS) like automation and engine controllers, avionics or medical devices, it is necessary to follow heavy-weight development processes (HWP) in order to satisfy safety development standards like IEC 61508[Bel05]. This is necessary to achieve certain *Safety Integrity Levels* which are the base for safety certifications. Among other things those standards define guidelines for planning, developing and testing the whole system and its single components. Those standardized procedures minimize safety critical bugs and errors in software but lead to extremely high development costs. Today a major goal of industrial software development units is to optimize and change their heavy-weight processes to reduce costs and time-to-market to compete economically. As a first step for cost reduction, many software development units have chosen C++ (for years C was the dominating language) as their preferred programming language for safety-critical hardware-dependent systems. Reasons for that are the inherent language flexibility, its potential for portability across a wide range of micro controllers and other hardware and, in contrast to native C, the possibility to use object-oriented language constructs for better code maintenance. A second step for optimization is to push the development processes towards agile processes[Mar03] which are

much more flexible and less cost intensive than HWP. For the introduction of new development processes in the area of safety-critical software it is yet indispensable to create new approaches for verification and validation to be in accordance with guidelines of standards like IEC 61508 to get safety certificates. Because the above mentioned systems are mostly event driven (waiting for a time tick, a signal of a sensor or a key press) this method aims at validating the correctness of a FSM which is common practice to handle events in software. The behavior of state machines can easily be described by UML state charts, even on a high abstraction level (in contrast to a formal FSM). An additional benefit of object-oriented languages is that a well-known design pattern exists to implement state machines. Nevertheless, we are in need of a methodology to validate such a state machine even at early stages. This is necessary because the state machine affects the whole software architecture and is responsible to put the system in a safe state when an error occurs. To conform to IEC 61508 it is necessary to test a system intensively with unit, system and integration tests. Therefore, often test cases are generated automatically. To estimate test quality, metrics like *percentage of covered lines of code* or *percentage of executed statements* are used. But this metrics can lead to a major misjudgment: Even high coverage values cannot promise that all critical areas of code have been tested (chapter 2.1 provides an overview of common coverage techniques). The main goal of this work is to create a methodology which presents more suitable metrics to estimate test quality: State-based coverage. During coverage analysis a lot of profiling information is generated (distribution of source code among files and directories). The results are presented in chapter 4. Furthermore, a behavioral check between UML state chart and C++ implementation is executed. An invalid transition taken during test execution can be revealed. Details about the presented validation approach can be found in chapter 3.

2 Preliminary Work and State of the Art

In this chapter we present state of the art coverage analysis techniques. In addition we comment patterns and language constructs to implement state machines using C++ and we classify the most important C++ software defect classes which are relevant for the design of SCHDS. Furthermore we discuss the Simulink/Stateflow[1] (SL/SF) approach and point out weak points of agile software developement for safety-critical systems.

2.1 Coverage Analysis

Coverage tools[YLW06] are commonly used to measure quality and completeness of test runs based on unit, system and integration tests[MS04]. These tools are counting how often code fragments (single lines of code, basic blocks or statements), branches and decisions are executed. Also the calling of methods and functions can be measured. Most coverage tools instrument the source code with so called probes which are injected into the

[1] Simulink and Stateflow are trademarks of The MathWorks Inc.

executable to be examined. This additional code tracks whether lines of code, branches or statements have been reached. Another possibility is to embed the executable into a monitor system which tracks the execution information from outside. The information gathered during a coverage analysis can be used to identify weak points in the source code: At first dead source code can be identified. These parts have never been executed or reached. Such code is just overhead and unnecessarily increases the size of source code files and compiled executables. Furthermore, test quality can be estimated, no matter what kind of test has been used. The more lines of code have been executed the better the unit test was. Table 1 presents an overview of available coverage methods an their corresponding metrics to describe test quality.

Coverage Method	Metrics	Complexity
Line Coverage	$Q = \frac{ExecutedLines}{AllLines}$	low
Statement Coverage	$Q = \frac{ExecutedStatements}{AllStatements}$	low
Branch Coverage	$Q = \frac{ReachedBranches}{AllBranches}$	medium
Path Coverage	$Q = \frac{TakenPathes}{AllPathes}$	high
Condition Coverage	All conditions must be executed with true & false	high

Table 1: Overview of coverage methods

For our methodology we are in need of a coverage tool which provides line coverage, a command line interface (CLI) and the ability to dump coverage data during execution (this is necessary to connect the source code with the single states). In Table 2 we present a feature overview of all C++ coverage tools taken into account: Bullseye[Bul09] is not able to perform a real line coverage. It is limited to functional and decision coverage. In contrast to Bullseye all other examined tools provide the feature of line coverage which is one of the main criteria for our concept. Decision coverage is also supported by CodeTEST[Met09], Dynamic[Sys09] and C++test[Par09] whereas functional coverage is only integrated in Bullseye, Dynamic and Intels code coverage tool[Int09]. All listed coverage tools bring along a file reporting feature. This means, that all coverage results are stored into files and are not only presented in a GUI (Bullseye, CodeTEST and C++test also support GUI-based reporting). Except Dynamic all coverage tools use a probe-based approach to measure coverage. These so called probes are small code fragments which are inserted into the executable during compile time. When a probe is reached and executed it increases the corresponding line or function counter. Dynamic does not instrument the code at compile time. Instead it uses a dynamic approach where the code is instrumented on the fly during execution time. File reporting is supported by all mentioned coverage tools. But the ability to dump coverage data into a file during execution is supported only by the open source tool Gcov[GNU09] which is part of the GNU compiler collection. That is the reason why we have chosen Gcov for our methodology.

Tool name	Line coverage	CLI	File	Dump during exec.	commercial	open source
Bullseye		✓	✓		✓	
CodeTEST	✓		✓		✓	
Dynamic	✓	✓	✓		✓	
Gcov	✓	✓	✓	✓		✓
Intel	✓	✓	✓		✓	
C++test	✓		✓		✓	

Table 2: Criteria for coverage tools to be suited for the presented methodology

2.2 Simulink Stateflow

Simulink Stateflow (SL/SF)[Mat09] is currently the de-facto standard in the area of model-based development. Simulink models consist of connected blocks which represent operations. Stateflow is able to describe the system behavior using parallel and hierarchical state machines as well as flowcharts. SL/SF provides the possibility of rapid prototyping and easy testing. A key feature of this tool bundle is the simulation of the whole model. A graphical debugger helps to reveal unexpected system behavior. Approaches like [KAI+09] and [AKRS08] optimize the symbolic analysis of traces and simulation coverage. SL/SF is able to generate C code but a major drawback is the lack of C++ code generation: Object-oriented languages provide more modern constructs and patterns to describe and dispatch finite state machines (see chapter 2.5) which often eases system integration. Furthermore, the developer has the possibility to insert hand-written code into SL/SF models which makes a subsequent validation or verification indispensable.

2.3 Agile Methods and Heavy-Weight Plan-Driven Processes

Agile processes have gained tremendous acceptance in industrial software development over the past years. A lot of research projects have examined the quality assurance abilities (QAA) in agile processes to test their relevance for development of safety-critical systems[HVZB04]. It is hard to compare agile methods with plan-driven strategies[Boe02] due to the big differences in costs and team size (agile methods have a smaller team size and therefore lower costs and also usually deal with more restricted system complexities). Therefore other approaches try to combine agile and plan-driven development processes[BT03] to take advantage of their strengths. We believe that it is possible to evolve agile processes such that they will become suitable for industrial light-weight development of safety-related systems. But it is necessary to provide the right tools for cost-efficient early-stage validation of software components. Our presented methodology (see chapter 3) contributes to that topic.

2.4 Software Defect Classes

Software defects can be categorized into several different classes. Table 3 presents an overview of some possible software defects in C++ software[MIS08] for safety-related systems. In addition we show which errors can be avoided by using our FSM behavioral checker (FBC).

Defect Class	Solution Strategy
Mistakes in specification (state machine)	Hugo/RT
Misunderstanding the specification or programming mistakes	FBC
Misunderstanding the language; compiler errors; runtime errors	Misra-C++, FBC

Table 3: Overview of software defect classes and solutions to avoid these errors

To detect, clasify and track software errors, a failure mode and effects analysis [PA02] (FMEA) can be integrated into the development process. A few software defects occurring in C++ programs can be avoided by sticking to the guidelines of MISRA-C++[MIS08]. But the defect classes mentioned in Table 3 need some other solution concepts.

- **Mistakes in specification:** These mistakes, e.g. a transition which points to a wrong target state and therefore create deadlocks, could be avoided by using a model checker like Hugo/RT to check state charts defined in UML.[BBK+04].
- **Developer misunderstands the specification or makes mistakes:** When the developer team misunderstands the specification, for example the team misinterprets the modeled behavior or functionality of the state diagram, behavioral errors in the implementation will occur. It is also possible that, despite correct understanding, implementation mistakes concerning the general behavior of the software component are made. To detect these bugs FBC can be used. FBC can detect deviations between implementation and specification.
- **Misunderstanding the language; compiler errors; runtime errors:** The developer team may misunderstand the effects of some language constructs. There are a number of areas of the C++ language that are prone to developer-introduced errors. If the developers follow the Misra-C++ definitions, those errors can be avoided. There are some areas in the C++ language that are not completely defined. The behavior varies from one compiler to the other. Even the behavior can vary within the same compiler, depending on the context. Misra reduces the C++ language to a proven subset. Following these guidelines compiler errors can be reduced, but Misra-C++ can not ensure a correct compilation process. In addition C++ programs tends to be small and efficient but they have a lack of data-dependent runtime checks. To deal with these defects our FBC can come into action.

2.5 State Machine Implementations

In this chapter we discuss four common ways to implement state machines in C++[Sam08].

1. A **nested switch statement** is perhaps the most simple FSM implementation. It consists of enumerated transitions and states and only one state variable is necessary. But it does not promote code reuse (here it breaks with agile processes) and it can easily grow too large. It is also very error prone because there is no language construct to control whether a valid transitions was taken or not.

2. A **generic state-table event processor** can be used to implement a FSM based on a simple two-dimensional state table. This table lists events along the horizontal and states along the vertical dimension. Therefore all states are listed in a regular and easy to read data structure. Also we only need a simple dispatcher which maps events to source and target states and checks if they are valid or not. Another advantage is the dispatcher can be used for more than one software projects (This is more conformant to agile methods than the *nested switch statements*).

3. The **state design pattern** is the object-oriented approach to implement a state machine. It relies heavily on polymorphism and partitions the single states in different classes. Transitions are efficient because only a pointer must be reassigned. All states are derived from an abstract super state class. The performance is better than indexing in a *state table*. But the implementation is not generic and the code cannot be reused for other state machines.

4. A **single object instance** of a class can be described by a FSM. Methods represent the triggers and attributes represent the states. More than one state encoding variable is possible.

3 The Methodology

Our methodology faces two major challenges. At first we want to adjust the abstraction level of specification and implementation for coverage analysis data: Both become state-based. The second aspect is to detect behavioral deviations between specification and the corresponding object instance during runtime. As a side-product many profiling information occur. In this chapter we present our methodology and its corresponding implementation in terms of a C++ library. At first we explain the main stages of our approach (chapter 3.1). Afterwards we go into detail (chapter 3.2) by presenting the industrial automation infrastructure software which is used later on in chapter 4 for an empirical study.

3.1 Main Stages

Our methodology is divided into four main stages (see Figure 1): Input, instrumentation, runtime and coverage stage.

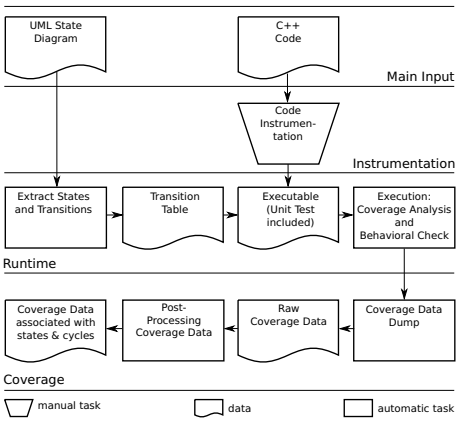


Figure 1: Flowchart of the methodology

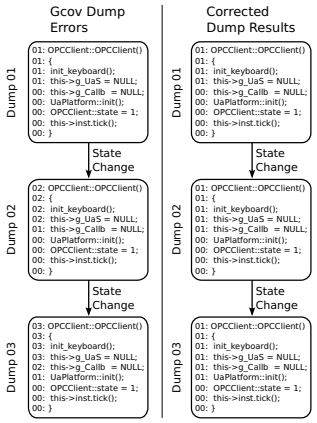


Figure 2: Post-processing of the coverage data

1. **Input stage:** Our methodology checks a state chart against its corresponding C++ implementation and associates the states with the corresponding source code. It is necessary to provide a *Class under Test* (CUT) and a UML state chart represented in XMI[2] (see Figure 1, Main Input).

2. **Instrumentation stage:** To apply our methodology to a C++ class it is important that the states of the FSM are encoded by one or more class members (attributes). Without this explicit encoding of states it is not possible to track the behavior of the object instance. Right now, our approach is based on a manual source code instrumentation. That means, we have to register all state encoding variables for monitoring and we have to integrate a timing reference in terms of a *tick()*-method to trigger the FSM (see Figure 1, Instrumentation).

3. **Runtime stage:** During runtime our library reads in the XMI file and creates a transition table. Based on this table our methodology is able to check whether a correct transition is taken or not. The CUT has to be linked against the library (see Figure 1, Runtime).

4. **Coverage stage:** During execution the behavior is checked (are only valid transitions taken?) and coverage data is dumped and stored in temporary files. Afterwards the coverage data is post-processed and connected with the corresponding states of the FSM (see Figure 1, Coverage).

[2]The most common use of XMI is as an XML-based interchange format for UML models

3.2 Details based on Industrial Automation Infrastructure Software

In this chapter we present the details of our approach by means of an industrial automation infrastructure software[MLD09] (IAIS). OPC UA consists of a client-server implementation (the focus in our experiments is on the client) which is used to access field data from field devices. In this example the client is used to monitor temperature data from an engine controller. The client connects to the server and is then able to receive data of the engine controller either with single reads or based on a subscription service which will deliver data in regular intervals. Furthermore, the client is able to browse the attributes provided by the engine controller and can also write data to the server. It consists of seven states which can be seen in Figure 3. Each state in the pseudo state (we want to keep the figure clear) can be reached by calling one of the following methods: *browse()*, *read()*, *write()* or *subscribe()*. The transitions are labeled with precise method names, including the return type, of the client class. Each call of one of these methods triggers the FSM to switch to the correct target state. The state encoding relies on the variable *int state*, written before a tick is executed. Of course the state encoding can rely on more than one variable. But to keep the example clear, we have chosen only one. The lines 2 and 9 in Figure 4 show the additional code caused by the necessary manual instrumentation. Figure 6 shows a small test scenario to stimulate our client without an error: All transitions are triggered in correct order. In contrast to this Figure 5 shows a test case which leads to an error: The transition *int OPCClient::shutdown()* is called before the transition *OPCClient::disconnect()* was executed. For our experiments we have embedded the client into the well-known Cppunit framework[Ope09]. Figure 3 shows the graphical representation of the error case (the dashed transition).

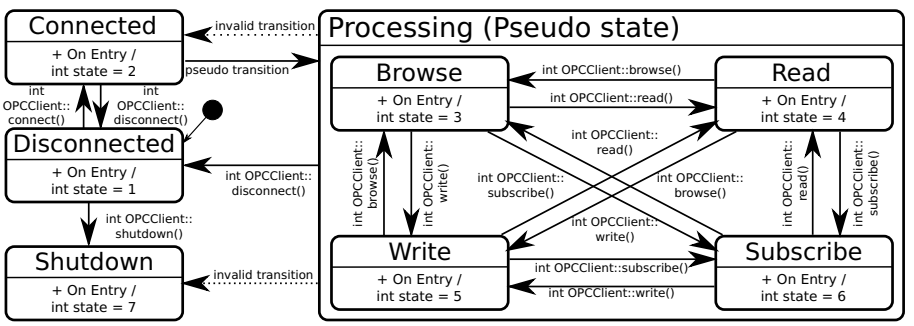


Figure 3: UML State machine representing the client behavior including invalid transitions (dashed arrows). Self transitions are valid for all states in the pseudo state. They have been omitted due to the lack of space.

To apply our validation approach to the client we have modeled the UML state diagram as seen in Figure 3 using Enterprise Architect (EA) in version 7.5[Spa09]. After that the FSM has been exported into a XMI file which must be stored in a directory together with all instrumented C++ source code files. An example of an instrumented file can be seen in Figure 4. In the default constructor the state encoding variable *state* must be registered for monitoring and the *tick()*-method must be called after each write access of this variable.

These ticks are used as a time reference for the FSM. Now the source code is ready to be compiled. After a successful compilation the CUT can be executed. During execution, the coverage data is dumped into a directory specified during the installation process of our library[3]. After a complete run the post-processing script is automatically triggered. This script processes all coverage data and presents a report as an HTML document. The flow sheet in Figure 1 provides an overview of the whole process. Following, we describe the single steps of the process in detail.

```
1  OPCClient::OPCClient() {
2    this->inst.regMonitorVar("state",
3                             "int",
4                             &this->state);
5    OPCClient::state = '1';
6  }
7  int OPCClient::browse() {
8    // browse the OPC Server
9    OPCClient::state = 3;
10   this->inst.tick(BOOST_CURRENT_FUNCTION);
11 }
```

Figure 4: Instrumented class Door: Default constructor and the *close()*-method

```
1  #include "OPCClient.hpp"
2
3  int main()
4  {
5    OPCClient client;
6    client.connect();
7    client.browse();
8    client.read();
9    client.shutdown();
10   return 0;
11 }
```

Figure 5: Unit test causing an error: An invalid transition is taken

```
1  #include "OPCClient.hpp"
2
3  int main()
4  {
5    OPCClient client;
6    client.connect();
7    client.browse();
8    client.disconnect();
9    client.shutdown();
10   return 0;
11 }
```

Figure 6: Unit test causing no errors: Only valid transitions are taken

- **UML State Diagram:** Modeled with EA the UML state diagram must follow some special design guidelines. Each transition must be labeled with the exact (return type, parameters) method name which triggers the corresponding transition. Each state must include one or more state variables representing the state encoding. These variables have to be modeled as an entry action. At the end the state diagram has to be exported into an XMI file.
- **Extract States and Transitions:** An XML parser reads in the XMI file and extracts all information of the modeled FSM.
- **Transition Table:** Based on the information obtained from the XMI file, the FBC creates a transition table. This table represents the core data of the behavioral checker. After triggering the state machine, our library checks if the transition as well as target and source state are valid.
- **C++ Code:** The C++ Code is the implementation of the FSM which we want to check (the CUT).
- **Code Instrumentation:** The state encoding variables have to be registered and the *tick()*-method has to be injected after each write access of these variables (see Figure 4 line 2).
- **Executable:** The CUT and its corresponding unit test have to be compiled with special compiler flags. We have chosen Gcov as our coverage tool. To make it work it is important to use the *-fprofile-arcs -ftest-coverage* flags supported by the g++

[3]We cannot use command line parameters because we have to avoid collisions with command line parameter needed by the CUT.

compiler. These flags control the code instrumentation for our coverage analysis based on Gcov. The manual code instrumentation mentioned above is only needed for the behavior check.

- **Execution:** The compiled source code can now be executed. Our library checks whether the whole course is correct or not (deviation detection).
- **Coverage Data Dump:** After each tick, the line coverage data is dumped into separate directories and is associated with the single states.
- **Raw Coverage Data:** The coverage data is distributed among subdirectories representing the sates.
- **Post-Processing Coverage Data:** After a complete execution of the CUT the library has to recalculate the coverage values. This correction is needed because Gcov adds up all line numbers of each dump (Gcov was not designed for dumping data during runtime). This leads to incorrect coverage values. Figure 2 shows two state changes and the line counter variance (the first column in the Figure). The second part of the Figure shows the recalculated line coverage values.
- **Coverage Data associated with States and Cycles:** At the end we receive an detailed graphical overview of all coverage results in HTML (the HTML generation is based on genhtml, a simple conversion tool provided with Gcov).

4 Results

We have successfully applied our methodology to the IAIS client presented in chapter 3.2. In this chapter we discuss the achieved results of a test run which has reached 9 states of the FSM. At first we have measured the runtime to show the scalability and overhead of our approach. Figure 7 shows the execution time of four test runs with 3, 5, 9 and 15 taken transitions and reached states (each valid taken transition triggers a data dump). We can see a constant growth relative to the number of reached states. Therefore our approach scales linear. Due to the heavy file I/O-operations caused by the coverage data correction the major part of computational time is spent for the post-processing of this data (see Figure 8. Each included source file is stored in a separate Gcov HTML file (generated with genhtml). In our test run with 9 reached states and therefore 9 taken transitions we have to parse 46 files per state. In addition to that we also have to correct the overview files produced by genhtml: One per included directory, in total 11 (see Table 4) and one for the whole overview. All in all our library had to post-process 453 files.

In our next two experiments we have provoked errors to test the behavioral checking feature of our library. In the first test run the *OPCClient::shutdown()*-method was called before the client has been disconnected from he server. In the second run we have called the *OPCClient::connect()*-method while one of the sub-states of the pseudo state has been active. Both errors have been detected correctly and the corresponding error traces have

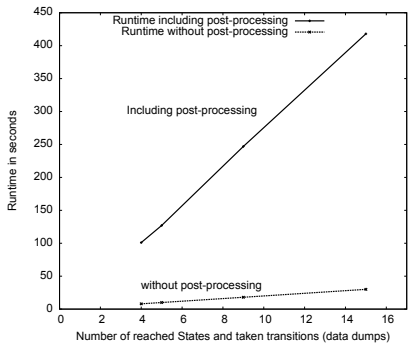


Figure 7: Runtime of 4, 5, 9 and 15 reached states: Constant growth

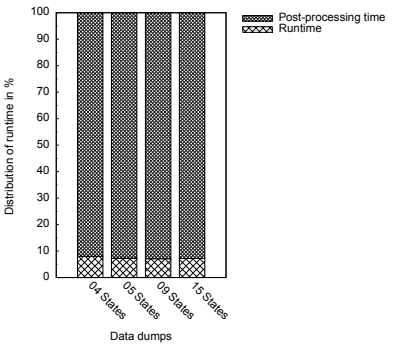


Figure 8: Runtime ratio of simulation time and post-processing time

been created. These traces include the execution order of all states until the error occurs. These traces are important for reproducing errors during the development process.

Directory Name	Disconnect (1)	Disconnect (2)
/usr/include/c++/4.3	3.03 %	0.0 %
/usr/include/c++/4.3/backward	28.57 %	0.0 %
/usr/include/c++/4.3/bits	20.73 %	0.0 %
/usr/include/c++/4.3/ext	4.35 %	0.0 %
/usr/include/c++/4.3/i486-linux-gnu/bits	0.0 %	0.0 %
/usr/local/include/cppunit	3.13 %	15.63 %
/usr/local/include/cppunit/extensions	92.00 %	0.0 %
examples/OPCClient/unittest	5.9 %	5.65 %
examples/utilities/linux	33.33%	0.0 %
include/uabase	7.14 %	45.24 %
include/uaclient	0.0 %	13.73 %

Table 4: Covered lines of code per directory in percent of the first and second hit of the state *disconnected*: The profile of the hits differ significantly.

Because the OPCClient implementation includes a huge number of source code files we cannot present the whole source code connected with its corresponding states. As an example we present an extraction of the file OPCClient.cpp including the main part of the client implementation. The code fragment can be seen in Figure 10. It presents the source code connected with the disconnect state when it is run through for the second time. Line 138 is marked as not covered. The reason for this is, that the tick()-method must be called before the return statement is executed. In chapter 5 we propose a method to avoid instrumentation and therefore lead to a correct coverage result by connecting also the return statement with the corresponding state.

In Figure 9 we present an overview of executed source code lines per state. Furthermore this figure shows, that all states has been covered. With metrics $Q = \frac{ReachedStates}{AllStates}$ we provide a formula to estimate test quality. In our test run we have achieved a state coverage

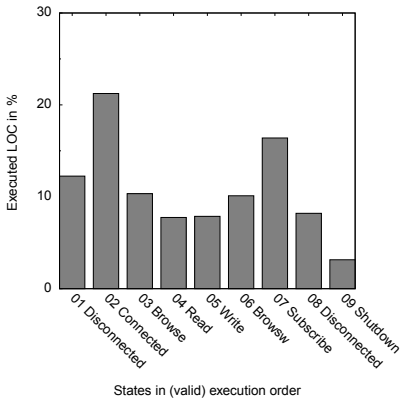


Figure 9: Executed code per state in percent

```
115| 1: int OPCClient::disconnect() {
116| 1:   printf("** Disconnect from Server\n");
117| 1:   if ( this->g_pUaSession == NULL ) {
118| 0:     printf("** Error: Server not connected\n");
119| 0:     return 1;
120|  :   }
121|  :
122| 1:   ServiceSettings serviceSettings;
123|  :   this->g_pUaSession->disconnect(
124|  :                       serviceSettings, // Use default settings
125| 1:                       OpcUa_True);     // Delete subscriptions
126|  :
127| 1:   delete this->g_pUaSession;
128| 1:   this->g_pUaSession = NULL;
129|  :
130| 1:   this->g_VariableNodeIds.clear();
131| 1:   this->g_WriteVariableNodeIds.clear();
132| 1:   this->g_ObjectNodeIds.clear();
133| 1:   this->g_MethodNodeIds.clear();
134|  :
135| 1:   OPCClient::state = 1;
136| 1:   this->inst.tick(BOOST_CURRENT_FUNCTION);
137|  :
138| 0:   return 0;
139|  : }
```

Figure 10: The *disconnect()*-method in *OPCClient.cpp* reached the second time (state *disconnected*)

of 100 %. We have shown that our test case has executed all safety relevant states. This will avoid the misjudgment of test quality based on common metrics presented in chapter 2.1.

5 Conclusion and Future Work

In this paper we have presented a methodology to connect states of a UML state chart, specifying the behavior of a C++ class, with the corresponding lines of code of all involved source files. This adjusts the abstraction level of specification and implementation (both are state-based now) and makes it much easier to estimate quality of test scenarios. Furthermore, our library is able to detect behavioral deviations between CUT and the state chart. Moreover we have successfully applied our library to an industrial automation infrastructure software. We have shown that the approach scales very well and that it has the potential to be a vital component to push HWP towards agile software development processes.
To simplify the use of our library we want to introduce a monitor mechanism, based on the GNU Debugger[SPS02] and Valgrind[NS07], which is able to observe trigger method calls and write accesses for state encoding variables. This will avoid the task of manual source code instrumentation. As a second extension we want to integrate a mechanism to execute all invalid (not available) transitions to perform stress tests.

6 Acknowledgments

We would like to thank ABB Corporate Research for co-funding our research and providing an industrial software to test our approach.

References

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98, New York, NY, USA, 2008. ACM.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In *ICFEM*, pages 434–448, 2004.

[Bel05] Ron Bell. Introduction to IEC 61508. In Tony Cant, editor, *Tenth Australian Workshop on Safety-Related Programmable Systems (SCS 2005)*, volume 55 of *CRPIT*, pages 3–12, Sydney, Australia, 2005. ACS.

[Boe02] Barry Boehm. Get Ready for Agile Methods, with Care. *Computer*, 35:64–69, 2002.

[BT03] Barry Boehm and Richard Turner. Using Risk to Balance Agile and Plan-Driven Methods. *Computer*, 36(6):57–66, 2003.

[Bul09] Bullseye. Bullseye Coverage, 11 2009. http://www.bullseye.com/.

[GNU09] GNU. Gcov Coverage, 11 2009. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[HVZB04] Ming Huo, June Verner, Liming Zhu, and Muhammad Ali Babar. Software Quality and Agile Methods. volume 1, pages 520–525, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[Int09] Intel. Intel Code Coverage Tool, 11 2009. http://www.intel.com/cd/software/products/asmo-na/eng/219633.htm.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivančić, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 430–445, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[Mat09] MathWorks Inc. http://www.mathworks.com/products/stateflow/, 07 2009.

[Met09] Metrowerks. CodeTEST, 11 2009. http://www.metrowerks.com/MW/Develop/AMC/CodeTEST/default.htm.

[MIS08] MISRA. *MISRA-C++:2008 Guidelines for the use of the C++ language in critical systems*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2008.

[MLD09] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer Publishing Company, Incorporated, 2009.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[Ope09] Open Source Community. CPPUnit, 11 2009. http://sourceforge.net/projects/cppunit/.

[PA02] Haapanen Pentti and Helminen Atte. Failure Mode and Effects Analysis of software-based automation systems. In *VTT Industrial Systems, STUK-YTO-TR 190*, page 190, 2002.

[Par09] Parasoft. C++TEST, 11 2009. http://www.parasoft.com/.

[Sam08] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, Newton, MA, USA, 2008.

[Spa09] Sparx Systems. Sparx Systems - Enterprise Architect, 11 2009. http://www.sparxsystems.de/.

[SPS02] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9 edition, 1 2002.

[Sys09] Dynamic Memory Systems. Dynamic Code Coverage, 11 2009. http://dynamic-memory.com/.

[YLW06] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, New York, NY, USA, 2006. ACM.

Improved Underspecification for Model-based Testing in Agile Development

David Faragó (farago@kit.edu)
Karlsruhe Institute of Technology, Institute for Theoretical Computer Science

Abstract: Since model-based testing (MBT) and agile development are two major approaches to increase the quality of software, this paper considers their combination. After motivating that strongly integrating both is the most fruitful, the demands on MBT for this integration are investigated: The model must be underspecifyable and iteratively refineable and test generation must efficiently handle this. The theoretical basis and an example for such models is given. Thereafter, a new method for MBT is introduced, which can handle this more efficiently, i.e., can better cope with nondeterminism and also has better guidance in the model traversal. Hence it can be used in agile development, select more revealing tests and achieve higher coverage and reproducibility.

key words: model-based testing; agile development; iterative refinement; nondeterminism; ioco; on-the-fly; off-the-fly;

1 Introduction

Model-based testing (MBT) and agile development (AD) are two main approaches to overcome the difficulties of reaching high quality in complex, ubiquitous software. As AD's strength is validation and MBT's strength is verification, we investigate how they can benefit from one another.

Section 2 shortly introduce AD, describe its deficits and its implications on verification. Section 3 introduces MBT with an underlying conformance testing theory as suitable formal method for this verification. Thereafter, the state of the art of MBT tools is described. Section 4 introduces related work and then shows the benefits of strongly integrating MBT and AD, as well as its requirements. In the last subsection, the requirements on the specification are solved by introducing Symbolic Transition Systems (STSs) and their underspecification and refinement possibilities. The end of the subsection concludes how MBT and AD are technically combined. Section 5 describes improved techniques for MBT in this combination with AD. This is our currently researched method, which efficiently processes the underspecified models and has further advantages. The conclusion gives a summary and future work.

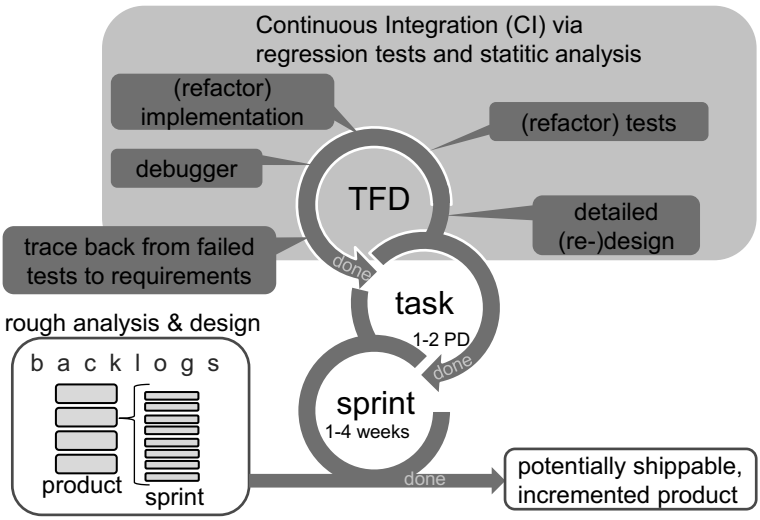


Figure 1: Exemplary agile method: XP and Scrum

2 Agile Development

2.1 Introduction

This section will briefly describe the main techniques of AD (cf. [SW07]), Subsection 2.2 AD's deficits and 2.3 the implications on verification.

In short, AD is iterative software development, such that requirements and solutions can evolve. This is supported by a set of engineering best practices, such that high-quality software increments can be delivered rapidly.

The big picture on how AD aims at better software development is given by AD's *values*, stated in the *Manifesto for Agile Software Development* (see [FH01]):

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

Figure 1 sketches an example of how software development achieves these values. Two of the most prominent agile methods are used, which can be combined easily: Extreme Programming (XP) [Tea98, JAHL00] and Scrum [TN86]. Other agile methods (e.g., the Agile Unified Process [Amb02] or Feature Driven Development [PF02]) lead to the same implications on verification.

AD achieves rapid delivery (see value 2) by short (a few weeks) development iterations (often called *sprints*): In each sprint, the team implements one ore more features from

the product backlog. These are often formulated with *user stories*, which are light-weight requirements - a few sentences in natural language. The user stories are broken down into tasks, which are put into the sprint backlog. By iterating over all tasks, the sprint is implemented. The result is a potentially shippable, incremented product.

In more detail, each task should be completable within 1 or 2 person days. Within a task, the developer practices even shorter iterations using *test-first development (TFD)*: after refining (or refactoring) the design, according test cases are specified (or refactored). Only thereafter the feature is implemented, using the IDE's semi-automatic features, such as method creation and refactoring. For more complex test case failures, the developer can use debugging and trace back from the test case to the according feature.

To assure value 2 in spite of flexibly being able to respond to change, AD practices *continuous integration* (*CI*), i.e., controlling the quality continuously. This is performed automatically using a CI framework in the background, e.g., *Hudson* or *CruiseControl*, which can use automated *regression tests*. AD's main focus on unit tests and acceptance tests (cf. [Rai09]). If less modular software is developed, more tests have to be performed by integration instead of unit tests. Unfortunately, this is often the case in practice. The CI framework can also use static analysis, e.g., to detect code smells and too high cyclomatic complexity. The team defines exit criteria, e.g., when tasks and sprints are done. These *definitions of done* are then given to the CI framework to be checked automatically.

Using agile processes, rapid delivery of high quality software increments can be achieved. These can be shown to the customer and her feedback can be flexibly incorporated in the following iterations. Hence AD is strong on validation. Therefore, a large fraction of organizations have adopted AD: two out of three according to the study [Amb08] together with Dr. Dobb's magazine, one out of three according to [For09]. Furthermore, the *Agile Conference* had a growth of 40% in 2009.

2.2 Deficits of Agile Development

The main deficit of AD is that too little specification and documentation is delivered. Specifications are becoming more and more important, e.g., for certificates and in todays component-oriented software development, since components need to be specified to reuse and distribute them. But more precise documentation is also needed by developers so they have direction and knowledge of the purpose while navigating through code, e.g., in pair programming.

AD also has some difficulties in testing, which is an integral part of AD (cf. the previous subsection): The used test coverage is often insufficient, e.g., 60%, and deceptive, e.g., statement coverage (cf. [LCBR05]). . Directly written test cases are less flexible and require more maintenance than the specifications for MBT. For instance, if exception handling is refined (as with the concise changes from Figure 2c to Figure 2d), a lot of test cases might have to be modified to incorporate this. Finally, tracing back from failed test cases to high level requirements is often difficult in AD.

Before the next section will show how MBT can help to overcome these deficits in AD,

the following subsection will describe in general the implications of AD on verification.

2.3 Verification

All agile processes require the following, which is relevant for verification: Firstly, being *flexible*, as result from value 1, 3 and particularly 4. Secondly, avoiding a big design up front (*BDUF*) by *rapidly delivering* working software, as result from value 2.

So specification should not obstruct AD's frequent iterations and support flexibility. To be able to integrate the verification process into the CI, it should be automatic and fast, such that developers get quick feedback. Optimally, a *10-minute build* (including CI) should be reached.

Formal methods (FM) can be used for the verification in AD to increase the degree of automation (e.g., using static analysis or model-based testing), the quality of the software, as well as the confidence in the quality. These aspects are especially important in safety-critical domains.The combination of AD and safety-critical software is being investigated in the project *AGILE* (see [Ope09]). It started in 2009 and firstly considers DO178B certifications for confidence in the quality, but other certifications will follow.

Investigating the combination of FM and AD has started some years ago, [BBB+09] gives a good overview.

J.B. Rainsberger stated on the *Agile Conference 2009* (see [Rai09]) that contract-based testing should replace integration tests in agile development. Several languages and tools exist for this, for instance JMLUnit [ZN10] and the MBT tool Spec Explorer [VCG+08].

Because of the above requirements on verification, light-weight formal methods are suited best. An example is the tool *FindBugs*, which is a static analysis tool on bytecode-level that detects certain bug patterns [APM+07]. It can also be plugged into Hudson. Unfortunately, it produces many false negatives.

MBT is a light-weight formal method that generates tests from specifications. Hence the whole system is still checked, and the formal method can easily be integrated into the development process - technically as well as psychologically, since agile teams are accustomed to testing. Hence this paper focuses on MBT.

3 Model-based Testing

3.1 Introduction

MBT can be used for conformance testing, i.e., to automatically check the conformance between a specification and the *system under test* (*SUT*), which is formally described in Subsection 3.2. To avoid error-prone redundancy and additional work, this paper considers MBT that automatically generates tests from the product's specifications, i.e., no

additional, explicit test specifications are necessary. MBT is mainly a black box technique and can automate all kind of tests: unit, integration, system and acceptance tests. For unit tests, MBT's models must be sufficiently refined to give low-level details. For acceptance testing, requirements must be integrated into the model.

The specifications are mostly written in a process algebraic language which defines a model as a *Labelled Transition System* (*LTS*) of some kind ([BJK+05]). The MBT considered in this paper uses *model checking* algorithms to derive test cases from the specifications: Paths are traversed in the model defined by the specifications, and witnesses to a considered requirement p, usually formulated as temporal logic formula, are returned. These witnesses (counterexamples for $\neg p$) yield test sequences: The inputs on the paths are used to drive the SUT, the outputs to observe and evaluate whether the SUT behaves correctly, i.e., as *oracles*. That way, MBT automatically generates superior black-box conformance tests compared to traditional, laborious testing techniques.

3.2 Ioco

MBT methods can be formalized and compared using the *input output conformance (ioco) theory*, a tool-independent foundation for conformance testing. The *ioco* relation determines which SUTs conform to the specification. The specification is given as LTS L describing the input and output (*i/o*). More precisely, $L = \langle Q, L_I, L_U, T, q_0 \rangle \in LTS(L_I, L_U)$, i.e., labels describe inputs L_I, outputs L_U, *quiescence* δ (aka *suspension*) or the internal action τ, cf. [Tre08]. L_I, L_U and the states Q are non-empty countable sets, $q_0 \in Q$ is the initial state, and $T \subseteq Q \times (L_I \cup L_U \cup \{\delta, \tau\}) \times Q$ the transition relation. The *suspension traces* of L, $Straces(L)$, is the set of all paths of L, the labels of T^* (T's reflexive, transitive closure), but all τ removed. The SUT is considered as *input output transition system* $\in IOTS(L_I, L_U)$, i.e., an input-enabled LTS $\in LTS(L_I, L_U)$, meaning that all inputs are enabled in all reachable states. The relation $ioco \subseteq IOTS(L_I, L_U) \times LTS(L_I, L_U)$ is defined as follows: $i \; ioco \; s :\Leftrightarrow \forall \sigma \in Straces(s) : \;\; \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$, where $\text{out}(x \text{ after } \sigma)$ means all possible outputs (or δ) of x after executing σ.

This notion can be used in the test generation algorithm to derive a test suite from the specification to check the SUT for ioco: By traversing s and nondeterministically choosing amongst all controllable choices, but keeping the branchings that are uncontrollable choices the SUT can take nondeterministically, the ioco algorithm generates test cases which are themselve LTSs $\in IOTS(L_U, L_I)$: they are output-enabled, cycle-free, deterministic, free of τ, finite, singular input- or δ-enabled (i.e., all states have exactly one input or δ enabled), and the leafs are exactly the verdicts. These test cases are *sound*, i.e., they do not report false negatives with respect to the ioco relation. The test suite containing all possible test cases is *exhaustive*, i.e., for each SUT that is not ioco-correct it contains a failing test case. A test case is run by executing it synchronously in parallel with the SUT.

3.3 State of the Art

The techniques that existing MBT approaches implement can be divided into two types:

- *Off-the-fly MBT* (also called *offline MBT*): First generate the complete test suite using model checking and then test the SUT classically by executing the generated test suite.
- *On-the-fly MBT* (also called *online MBT*): Generate and execute tests strictly simultaneously by traversing the model and the SUT in lockstep.

The MBT community has realized the need to also consider nondeterministic SUTs. This is particularly important for complex, e.g. distributed, systems, where the tester does not have full control (for instance because of race conditions and exceptions). Specifications that also offer nondeterminism can cover this situation and also enable abstractions helpful for AD (cf. Section 4.3). How effective nondeterministic specifications can be processed depends on the technique employed.

3.3.1 Off-the-fly Model-based Testing

Off-the-fly MBT (also called offline) used for instance by the tool TGV (cf. [BJK+05, JJ05]), first generates all tests using model checking and then executes them classically. The strict separation of test generation and test execution has several deficiencies: The high costs for intermediate representation, dynamic adaptations to the test generation are not possible, and nondeterministic SUTs cannot be processed effectively.

3.3.2 On-the-fly Model-based Testing

On-the-fly MBT (also called online) is being applied, for instance, by TorX [TBR03], UPPAAL TRON [LMN04] and also Spec Explorer [VCG+08, BJK+05, UL07]. It uses the other extreme of generating and executing tests strictly simultaneously by traversing the model and the SUT in lockstep. This eliminates all above deficiencies:

- Overhead in time and memory can be reduced since we no longer require an intermediate representation, such as the costly representation of the specification in the test suite generated by the ioco algorithm.
- Dynamic information from test execution can be incorporated in other heuristics, especially guidance, so that the MBT process can adapt to the behavior of the SUT at runtime and therefore generate more revealing tests. For instance:
  - Investigating the part of the state space around faults already found is a promising strategy because faults tend to occur in cliques. For instance, the coverage criteria can be strengthened for components in which many faults were found.
  - In contrast, if we can identify the causality between faults, we can avoid pursuing consequential faults.
  - When generating a test sequence, we can try to discharge as many test goals as possible in each state. This can be too liberal a strategy if dynamic adaptations are not available. If they are, we can subsequently generate more tests to isolate

the function deviating from its requirement.

- On-the-fly MBT can also process nondeterministic SUTs by instrumenting them to track the actual choices they made at runtime.

The directly executed tests can also be recorded, yielding a classical test suite that can later be re-executed by common testing tools without formal methods. Being able to *reproduce* a test execution helps to check whether a fault has been fixed or to provide repeatable evidence of correctness. If the SUT is nondeterministic, recorded test suites no longer guarantee reproducibility. The main drawback of on-the-fly MBT, however, is its weak guidance used for test selection. A solution is considered in Section 5. It also enables stronger utilization of coverage metrics as guidance while traversing the model graph, so that newly generated tests really raise the coverage criterion. As formal specifications contain control flow, data and conditions, the same coverage metrics can be applied as on source code level, for instance statement, transition or MC/DC coverage. Which coverage criterion is best on the specification level varies.

4 Model-based Testing and Agile Development

4.1 Related Work

Some previous work on MBT with AD are: [UL07] scarcely considers using AD to improve MBT and also MBT within AD. It suggests MBT outside the AD team, i.e., not strongly integrated. [Puo08] aims to adapt MBT for AD and also shortly motivates using MBT within the AD team, but does not investigate in detail how to modify AD for fruitful integration, e.g., adjusting specifications and CI. It rather focuses on a case study, which empirically shows that abstraction is very important in MBT for AD. [KK06] uses a strict domain and a limited property language (weaker than the usual temporal logics). It uses very restricted models that are lists of exemplary paths.

[Rum06] gives a good overview of MBT when evolution (as in AD) is involved. It uses the same modelling language for the production system and the tests, but not the same models. As mentioned in Subsection 3.1, our kind of MBT generates tests from the product's specifications to reduce work and redundancy.

No other author (I know of) differentiatedly investigates the requirements on our kind of MBT for AD, vice versa, or on both integrated strongly (which therefore covers both verification and validation). Such investigations have been made in [Far10] and are described in the following subsection.

4.2 Strongly Integrated

4.2.1 MBT for AD

The general approach of integrating MBT and AD is changing the TFD cycle in Figure 1 (cf. Figure 3): CI then uses MBT for regression testing and metrics such as various specification and code coverages. Therefore, specifications instead of tests are written (or refactored) - so this is rather a *specification-first development (SFD)* cycle. These are read by the MBT tool. Developers can now trace back from failed test cases to the corresponding specification, which is easier than the original tracing back to the corresponding requirement. The specifications are additional deliverables of the development process. Using MBT also counters deceptive and insufficient coverage, low flexibility and high maintenance. Additionally, AD requires rapid delivery and continous integration with regression tests. Hence MBT can profitably be applied to AD: Efficient tests can be generated and executed automatically with an appropriate coverage. The test suite can be changed flexibly by modifying the concise models. This is especially important for configuration management and *acceptance test driven development* (*ATDD*, see [Hen08]), where automated acceptance tests are part of the definition of *done*.

Furthermore, advanced coverage criteria not only produce better tests, but also better measurements for quality management, e.g. for ISO 9001. For instance, andrena object's agile quality management ISIS (see [RKF08]) is state of the art and considers many relevant aspects: Test coverage is only one of 10 metrics and measured using EclEmma. But that only allows limited coverage criteria, namely basic blocks, lines, bytecode instructions, methods and types (cf. [ecl]). Since automated tests are a central quality criterion, especially in AD, and since [YL06] shows that more sophisticated coverage criteria (such as MC/DC) are more meaningful, MBT in AD can also improve agile quality management.

If we do not modify MBT itself, though, i.e., do not integrate MBT and AD strongly with one another, we have some discrepancies, e.g., the specification languages and the definition of *done* do not match, so CI is not very effective. Furthermore, the benefits described in the next section do not take effect. Additional specifications for MBT are required, and must be flexibly changable. Finally, MBT must work fast: during specification to avoid a BDUF and during verification for a quick CI.

4.2.2 AD for MBT

AD is so successful because, amongst others, it fixes time and cost, but can handle the scope and changing requirements flexibly. This is also important for MBT to avoid rigidness and a BDUF (cf. previous Subsection). For this, the iterative and incremental processes in AD can be applied on the specification-level:

- starting with very abstract models, to support flexibility,
- iteratively refining aspects of the model within sprints, for rapid delivery,
- using the refined specifications that are sufficiently detailed for MBT.

Furthermore, agile methods as pair programming, reviews, but also CI help find defects in the specifications early. Because of AD's strong validation, differences between the specification and the customers expectations are also fixed. Finally, clearer and more modular source code that is packaged in increments can improve MBT's efficiency.

Therefore, MBT profits from AD, but powerful underspecification is necessary (especially in the first iterations).

4.2.3 Conclusion

The last subsections have shown that AD can profit from MBT and vice versa, but without mutual adaptions, i.e., strong integration, both MBT and AD restrict each other. For the integration, a unified specification that offers flexible underspecification is necessary, which will be presented in the following subsection. Since current MBT tools cannot efficiently handle these (cf. Section 3), the next section will introduce a new MBT method for this.

4.3 Specifications

This subsection looks closer at the specification types used in MBT and AD, and how they can be unified since multiple unrelated specifications are unnecessary costly, redundant (i.e., contradicting the DRY principle, [HTC99]), and make strong integration of MBT and AD difficult. The arguments are similar to those used for *agile modeling* (cf. [Amb02]), which does not focus on testing, though.

In AD, most specifications are very light-weight and mainly used to describe customer requirements. Often user stories are used, which are too abstract to be understood on their own. They are great for communication, though, e.g. between customers and developers, which yields more detailed customer requirements. These are usually put in the form of an acceptance test suite with a framework as FitNesse. Such test suites are often more data-oriented than flow-based, and the maintenance of test scenarios is difficult, as is achieving high coverage.

In MBT, specifications must be sufficiently detailed for deriving a test suite that is revealing. They are behavioral descriptions in *UML statecharts* or something similar, e.g., Labelled Transition Systems or *Symbolic Transition Systems* (*STSs*) as depicted in Figure 2. These are very powerful, as they have precise semantics, i/o and location (i.e. model-) variables, and conditions in first order logic. The labels of the ioco theory (cf. Subsection 3.2) are lifted and now have the form: `{input,output,internal,` δ`}:name.[guard]{update}`, with `name` being a method name, `guard` a formula in first order logic and `update` a term over all location variables and the i/o variables of `name`. The precise semantics are given in [FTW06]. STSs can describe the behavior of the SUT and requirements on several levels of abstraction. The most abstract level (cf. Figure 2a) can be used for communication and to give an overview. Although an abstract STS is less intuitive than a user story, the higher precision is more important, especially in safety-critical domains. Abstraction is achieved via underspecification by:

- allowing many nondeterministic choices for the SUT, using nondeterminism in the specification or defining a real superset of outputs in a state, e.g., by defining abstracted oracles via relaxed conditions
- ignoring certain situations (e.g., hazards) by defining a real subset of inputs in a state

The level of abstraction could also be influenced by the mapping from abstract test sequences (paths of the model) to concrete test sequences (execution traces for the SUT), but to be able to formalize underspecification and refinement, we use the underspecification techniques within the specifications. Therefore, we have for specifications $s_1, s_2 \in LTS(L_I, L_U)$: s_1 *refines* $s_2 \ :\Leftrightarrow \forall \sigma \in Straces(s_1) \cap Straces(s_2) : \ \text{in}(s_1 \text{ after } \sigma) \supseteq \text{in}(s_2 \text{ after } \sigma) \ \wedge \ \text{out}(s_1 \text{ after } \sigma) \subseteq \text{out}(s_2 \text{ after } \sigma)$.
Extending the ioco relation to $LTS(L_I, L_U)^2$, this can be written as s_1 accepts more inputs than s_2 and $s_1 \ ioco_{Straces(s_1) \cap Straces(s_2)} \ s_2$. With this, the following implication in relation to the SUT i holds: $i \ ioco \ s_1 \implies i \ ioco \ s_2$. So using the most refined specification for MBT within CI also guarantees the correctness up to the most abstract specification (that replaced user stories).

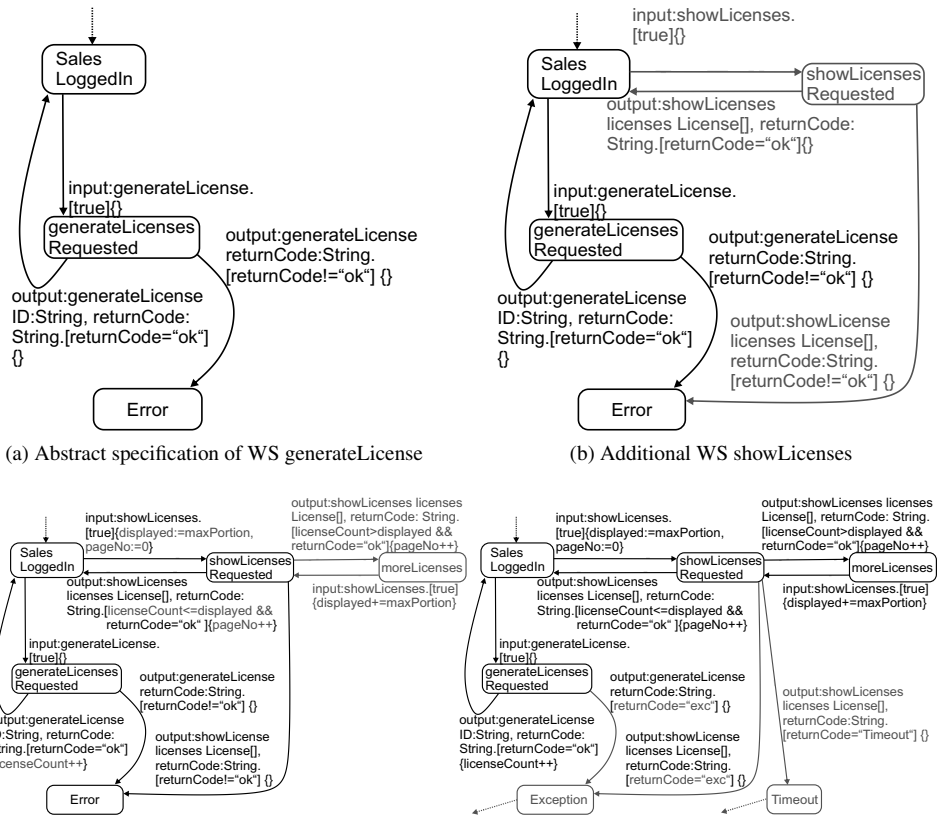
Figure 2 gives simplified exemplary specifications for web services generating licenses from WIBU SYSTEM AG's *License Central*, which is from the domain of *service-oriented architecture* (*SOA*). Such services can easily be tested via MBT and are frequently used in AD (and sometimes called *Agile Applications*), since their design concept supports AD: Services are simpler than monolithic systems and loosely coupled, assisting rapid delivery, fault tolerance and scalability.

Figure 2b refines Figure 2a by specifying more inputs. Figure 2c describes a different functionality than Figure 2b and therefore is not a refinement (state *moreLicenses* sometimes replaces state *loggedIn* but offers less inputs). Figure 2d refines Figure 2c by specifying less outputs.

All in all, the powerful specifications of MBT have the flexibility to be used for several purposes in AD (for business-facing as well as technology-facing, cf. [CG09]). They can particularly well replace the more precise models sometimes used in AD, e.g., UML statecharts and use cases (cf. [Coc00]). Those are used when technical details need to be considered early; e.g., when complex business or product logic, business processes or more generally complex architectures need to be analysed, described or implemented. The resulting workflow as extension to Figure 1 is depicted in Figure 3.

So prefering such a powerful specification over those used in AD leads to a unification with the following benefits:

- cost and redundancy are reduced
- AD can be applied to the refinement process of the models, i.e., when defining more detailed models by reducing the level of abstraction
- strong integration of MBT and AD is enabled, which is considered in the following sections.



(a) Abstract specification of WS generateLicense

(b) Additional WS showLicenses

(c) More functionality for showLicenses

(d) Refined exception handling

Figure 2: Exemplary STS specifications for web services generating licenses

5 Lazy On-the-fly Model-based Testing

5.1 Introduction

Using on-the-fly instead of off-the-fly MBT solves several deficiencies (cf. Subsection 3.3.2).

The main drawback of on-the-fly MBT, however, is its inability to use the backtracking capabilities of the model checking algorithms: Each step of the model traversal is executed strictly simultaneously in the SUT, which cannot undo these steps. The lack of backtracking complicates test selection since different subpaths (longer than one transition) cannot be chosen amongst. Instead, test selection has to be performed at the early stage of deciding which transition to traverse. All mentioned on-the-fly tools have the major disadvantage of weak guidance used for test selection.

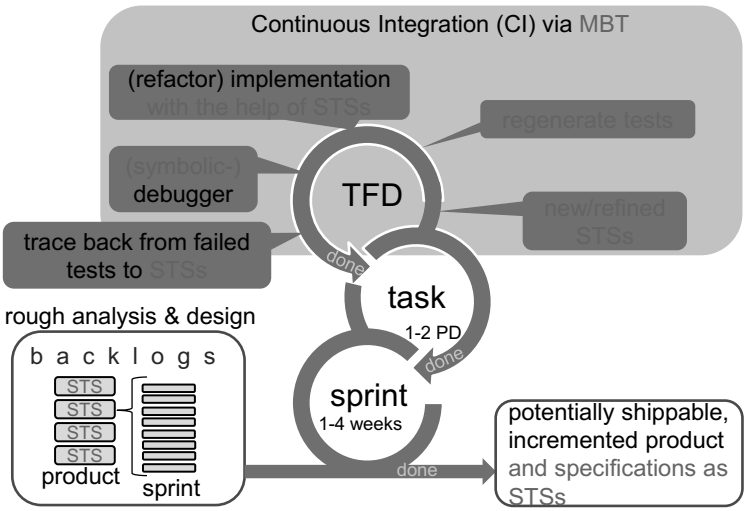


Figure 3: MBT with STSs in our exemplary agile method

To preserve the advantages of the two approaches and avoid their disadvantages, our project MOCHA[1] offers a novel method that synthesizes these contrary approaches: It aims at tackling this problem by designing a new method, *lazy on-the-fly MBT*, that fulfills ioco and can harness the advantages of both extremes, on-the-fly as well as off-the-fly MBT. It executes subpaths of the model lazily on the SUT, i.e., only when there is a reason to, e.g., when a test goal, a certain depth, an inquiry to the tester, or some nondeterministic choice of the SUT is reached, a so-called *inducing state*. Therefore, we do interleave model traversal and execution of the SUT, but not strictly after each test step, only loosely after several steps.

So the top-level algorithm for lazy on-the-fly MBT repeatedly compiles and executes subpaths in the following way:

1. $currentState :=$ initial state;
2. Traverse the subgraph containing all subpaths from $currentState$ to the next inducing states;
3. Select one subtree π;
4. Execute π in the SUT (as determined by the ioco theory) and log the results;
5. If the termination criteria are not yet met:
   if π ends with a terminal: GOTO 1;
   else $currentState :=$ last state of π; GOTO 2;

Lazy on-the-fly MBT can exploit the advantages of both extremes to a large extent:

- As in off-the-fly MBT, backtracking is again possible, now within the model's subgraphs that are bounded by the states where test execution is induced (*inducing*

[1] funded by Deutsche Forschungsgemeinschaft (DFG)

states). So from within a subgraph, we do not look backward or forward across inducing states, but we can search the complete subgraph for charged test goals, to choose the most promising subtree, e.g., the one passing the most charged test goals with the fewest possible steps. Hence lazy on-the-fly MBT is strongly guided by prioritizing subpaths or subtrees (*guidance on subpath scale*). This powerful guidance can easily incorporate other heuristics, enables new heuristics, harness dynamic information from already executed tests, e.g., nondeterministic coverage criteria, and therefore help generate and execute tests faster and more flexibly.

- The cost for local intermediate representations can strongly be bounded since only the current subtree is passed to the test driver controlling the SUT. If the SUT is nondeterministic and paths are pruned when nondeterministic choices are made, costly bifurcation is avoided.
- Nondeterministic SUTs can be processed easier, e.g., by inducing test execution at their nondeterministic choice points and using the execution result dynamically.

As result, we expect strong guidance to reduce the state space and to produce fewer and shorter tests with higher coverage, to reveal more relevant faults, for instance those which only occur after long sequences of events.

An example is testing `showLicenses` over multiple pages (cf. Figure 2d), which corresponds to the temporal formula $(\diamond pageNo > 1)$. This requires a test sequence that calls `generateLicense` more than `maxPortion` times and the `showLicenses` two times. Using off-the-fly MBT for this test does not work, since the variables in the STSs cause a potentially infinite state space and the nondeterministic choices that must be passed on the test sequence cause a huge test tree, e.g., for exception handling. Using on-the-fly MBT works better, but since it has weak guidance and the test sequence is so long, it is very inefficient: many `showLicense` calls (or other WS calls) will likely occur until `generateLicense` is called more than `maxPortion` times. With lazy on-the-fly MBT, we can choose error states and the test goal as inducing states and therefore get via better guidance a test tree that is very similar to the test sequence $\texttt{generateLicense}^{\texttt{maxPortion}+1}\texttt{showLicenses}^{2}$. It only has bifurcations of length one because after the alternative outputs for exceptions or timeouts, these paths are immediately pruned.

5.2 Within AD

The main advantages described in the previous subsection can improve the efficiency of testing, especially for abstract specifications and nondeterministic systems. This is particularly useful in AD, since underspecification supports AD, as it empowers flexibility and fast modelling for rapid delivery. Another advantage of lazy on-the-fly MBT is its strong guidance, which uses dynamic information. It efficiently finds short and revealing tests with better coverage criteria and helps reproducibility - also for nondeterministic systems. This is particularly important for automated regression testing in CI, which therefore performs continuous ioco checks. Finally, these coverage criteria can also improve measurements for quality management.

6 Conclusion

6.1 Summary

We investigated how MBT and AD can be combined: MBT for AD improves flexibility, maintenance and coverage. AD for MBT avoids rigid models and a BDUF. By unifying the specifications and strongly integrating MBT and AD, we get the highest profits with reduced cost and redundancy and with effective CI and sensible definition of *done* that continuously checks if the model is conform to the code. For this to work, we need abstract models. STSs on several abstraction levels are suitable. These can be processed more effectively with our new lazy on-the-fly MBT: It chooses partial execution at the most sensible point in time and can therefore cope with nondeterminism and leverage backtracking and dynamic information. This leads to better testing and reproducibility in AD.

6.2 Future Work

The main future work is implementing lazy on-the-fly MBT and conduction case studies.

Theoretical future work within MOCHA include modifying ioco for finer-grained refinements that are compatible with robustness tests, and identifying efficient heuristics that incorporate dynamic information, e.g., nondeterministic choices.

7 Acknowledgements

I would like to thank Peter H. Schmitt, chair of our Logic and Formal Methods Group, Leif Frenzel from andrena objects AG, and Alexander Schmitt, head of software development at WIBU-SYSTEMS AG.

References

[Amb02] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Amb08] Scott Ambler. Has agile peaked? *Dr. Dobb's*, 2008.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using FindBugs on production software. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 805–806. ACM, 2007.

[BBB+09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, 2009.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2005.

[CG09] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[ecl] EclEmma: Using the Coverage View. http://www.eclemma.org /userdoc/coverageview.html/. (April 2010).

[Far10] David Faragó. Model-based Testing in Agile Software Development. In *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility*, to appear in Softwaretechnik-Trends, 2010.

[FH01] M. Fowler and J. Highsmith. The Agile Manifesto. In Software Development, Issue on Agile Methodologies, `http://www.sdmagazine.com`, last accessed on March 8th, 2006, August 2001.

[For09] Agile Development Method Growing in Popularity. *Forrester*, 2009.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. A Symbolic Framework for Model-Based Testing. In Klaus Havelund, Manuel Nez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[Hen08] Elisabeth Hendrickson. Driving Development with Tests: ATDD and TDD. *STARWest 2008*, 2008.

[HTC99] Andrew Hunt, David Thomas, and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 1999.

[JAHL00] Ronald E. Jeffries, Ann Anderson, Chet Hendrickson, and Chapter Circle Of Life. Extreme Programming Installed, 2000.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[KK06] Mika Katara and Antti Kervinen. Making Model-Based Testing More Agile: A Use Case Driven Approach. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*. Springer, 2006.

[LCBR05] Joseph Lawrance, Steven Clarke, Margaret Burnett, and Gregg Rothermel. How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 53–60, Washington, DC, USA, 2005. IEEE Computer Society.

[LMN04] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems Using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2004.

[Ope09] Project AGILE, 2009.

[PF02] Stephen R. Palmer and John M. Felsing. *A Practical Guide to Feature-Driven Development (The Coad Series)*. Prentice Hall PTR, February 2002.

[Puo08] Olli-Pekka Puolitaival. Adapting model-based testing to agile context. *ESPOO2008*, 2008.

[Rai09] Joseph B. Rainsberger. Integration Tests Are a Scam. *Agile2009*, 2009.

[RKF08] Nicole Rauch, Eberhard Kuhn, and Holger Friedrich. Index-based Process and Software Quality Control in Agile Development Projects. *CompArch2008*, 2008.

[Rum06] Bernhard Rumpe. Agile Test-based Modeling. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 10–15. CSREA Press, 2006.

[SW07] James Shore and Shane Warden. *The art of agile development*. O'Reilly, 2007.

[TBR03] Jan Tretmans, Ed Brinksma, and Cote De Resyste. TorX: Automated Model Based Testing - Cte de Resyste, 2003.

[Tea98] The "C3 Team". Chrysler Goes to "Extremes". *Distributed Computing*, pages 24–26, 1998.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 1986.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. *Formal Methods and Testing*, pages 1–38, 2008.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 edition, 2007.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[YL06] Yuen-Tak Yu and Man Fai Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software*, 79(5):577–590, 2006.

[ZN10] Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The Next Generation. In *FoVeOOS 2010*, pages 20 – 29, 2010.

An Experience on Formal Analysis of a high-level graphical SOA Design

Maurice H. ter Beek Franco Mazzanti Aldi Sulova
ISTI–CNR, Via G. Moruzzi 1, 56124 Pisa, Italy
{terbeek,mazzanti,sulova}@isti.cnr.it

Abstract:
In this paper, we present the experience gained with the participation in a case study in which a novel high-level design language (UML4SOA) was used to produce a service-oriented system design, to be model checked with respect to the intended requirements and automatically translated into executable BPEL code.

This experience, beyond revealing several uncertainties in the language definition, and several flaws in the designed model, has been useful to better understand the hidden risks of apparently intuitive graphical designs, when these are not backed up by a precise and rigorous semantics.

The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow the full exploration of designs which otherwise risk to become simple to draw and update, but difficult to really understand in all their hidden ramifications. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using appropriate model transformation techniques. This is particularly valuable in the context of agile development approaches which are based on rapid and continuous updates of the system designs.

1 Introduction

Service-Oriented Computing (SOC) has emerged during the last decade as an evolutionary new paradigm for distributed and object-oriented computing [Pa07, SH05]. Services are autonomous, distributed, and platform-independent computational elements capable of solving specific tasks, ranging from answers to simple requests to complex business processes. Services need to be described, published, categorized, discovered, and then dynamically and loosely coupled in novel ways (composed, orchestrated) so as to create largely distributed, interoperable, and dynamic applications and business processes which span organizational boundaries as well as computing platforms. Their underlying infrastructures are called Service-Oriented Architectures (SOAs). Unlike any earlier computing paradigm, SOC is destined to exert a continuous influence on modern day domains like e-Commerce, e-Government, e-Health, and e-Learning.

We have actively participated in the IST-FET Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers), funded by the EU under the 6th framework programme's Global Computing initiative. SENSORIA has successfully developed a novel comprehensive approach to the engineering of service-oriented software

systems, in which foundational theories, techniques, and methods are fully integrated into a pragmatic software engineering process. We refer to [WH10] and the references therein for details on SENSORIA.

Within the context of SENSORIA the novel high-level graphical design language UML4SOA [Fo10, UML] has been defined with the aim of facilitating service-oriented system designs. The same notation has been used inside the project to specify the credit request scenario from the SENSORIA's Finance case study (see Sect. 2).

High-level graphical design notations are very intuitive and very efficient in describing the current structure and status of an ongoing software project. Especially if they are associated with automatic code generation / design verification features, they might play an interesting role inside an agile software development process as they can help in reducing the effort of evolution cycles, and they can help in maintaining the focus of the development at a level which is also understandable by the client. This might help the cooperation between the clients and the developers, promoting the rapid delivery of software and its regular adaption to the evolving requirements, in the spirit of the agile approach to software development [FH01, Mar02, SW07] .

The author's role in this case study was centered around the formal analysis of the original UML4SOA design. This goal required some kind of formalization of the UML4SOA semantics and the translation of the system design into a formal model suitable to undergo a model-checking phase. To this aim, we first translated the UML4SOA design of the credit request scenario by hand into a formal operational UMC model [BM10]. This provided us with many useful insights into an automatization of translating UML4SOA diagrams into UML statecharts, which finally resulted in a prototype of a translator (based on the standard Eclipse EMF format and its transformation capabilities). UMC [Be10, GM10, Ma09, UMC] is an on-the-fly model checker that allows the efficient verification of SocL formulae over a set of communicating UML state machines. The state machines are defined using the UML statecharts notation which has a standard presentation and semantics defined by the Object Management Group (OMG). SocL [GM10] is an event- and state-based, branching-time, efficiently verifiable, parametric temporal logic that was specifically designed to capture peculiar aspects of services.

Since the purpose of the case study was the just the experimentation with a novel design notation and with novel model transformation and verification approaches, in our case the software development process has been very informal and it did not have the rigorous structure of any kind of industrial software development method (whether agile or not).

Our experience, beyond revealing several uncertainties in the language definition, and several flaws in the designed model, has been useful to better understand the hidden risks of apparently intuitive graphical design notations, when these are not backed up by a precise and rigorous semantics; in this case, in fact, abstract designs risk to become simple to draw and update, but difficult to really understand in all their hidden ramifications. The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow to overcome the problem. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using appropriate model transformation techniques, and this is particularly valuable in the

context of agile development approaches which are based on rapid and continuous updates of the system designs.

This paper is organized as follows. In Sect. 2, we sketch the credit request scenario from SENSORIA's Finance case study. In Sect. 3, we briefly introduce the UML4SOA profile as developed in SENSORIA. Subsequently, we outline our translations from UML4SOA diagrams to UMC statecharts in Sect. 4 and in Sect. 5 . In Sect. 6 we present the lessons we learned from this formalization of UML4SOA designs into executable UMC code. Finally, we draw some conclusions in Sect. 7.

2 Finance Case Study: The Credit Request Scenario

The credit request scenario from SENSORIA's Finance case study was provided by one of SENSORIA's industrial partners, S&N [S&N], which is a leading IT company of the financial services industry. The scenario is specified in UML 2.0 by using the profiles SoaML [OMG], which defines a metamodel for designing the structural aspects of SOA, and UML4SOA [Fo10, UML], which defines a high-level domain-specific modeling language for behavioral service specifications (see Sect. 3).

The scenario's main services are CreditRequest and—to a lesser degree—Rating. These are complemented with web services CustomerManagement, SecurityAnalysis, BalanceAnalysis, and RatingCalculator that serve to interact with clients, save the information they provide, calculate ratings, and the like. The CreditRequest service describes a bank service that offers clients the possibility to ask for a loan and subsequently orchestrates the steps needed to process this request, which may involve an evaluation by a clerk or a supervisor before a contract proposal is sent to the client. The Credit Request service is depicted in Fig. 1. The scopes **Initialize** and **Finalize** handle a client's login and log off.

The loan workflow is represented in the scope **Main**, whose initial activity is to service the request. The **Creation** scope starts with a call from the Portal. The data involved is stored in the CreditManagement service and a confirmation is sent to the Portal. In case of a fault, the compensation handler removes the request from the CreditManagement service. The retrieval of the client's balances and securities is handled by the **HandleBalance&SecurityData** scope and these are stored in the Balance and Security services. Upon completion of the request, the Rating service calculates the rating of the client's request after being asked to do so through the **RatingCalculation** scope.

The Rating service calculates a rating, which implies whether the request can be accepted automatically or a clerk or a supervisor needs to decide this. The **Decision** scope takes care of this. Rating AAA automatically leads to an offer to the client. In any other case, the **Approval** scope is used for a decision by an employee, based on which an offer or a decline is generated, according to the **Accept** and **Decline** scopes. Rating BBB means that the request has some risk, but can be decided by a clerk, while CCC indicates a much more risky request that needs a supervisor to decide. The decision is sent to the Portal. An offer is saved in the CreditManagement service and sent to the Portal, allowing the client to see it and decide to accept or decline it through interaction with the Portal. In case of a

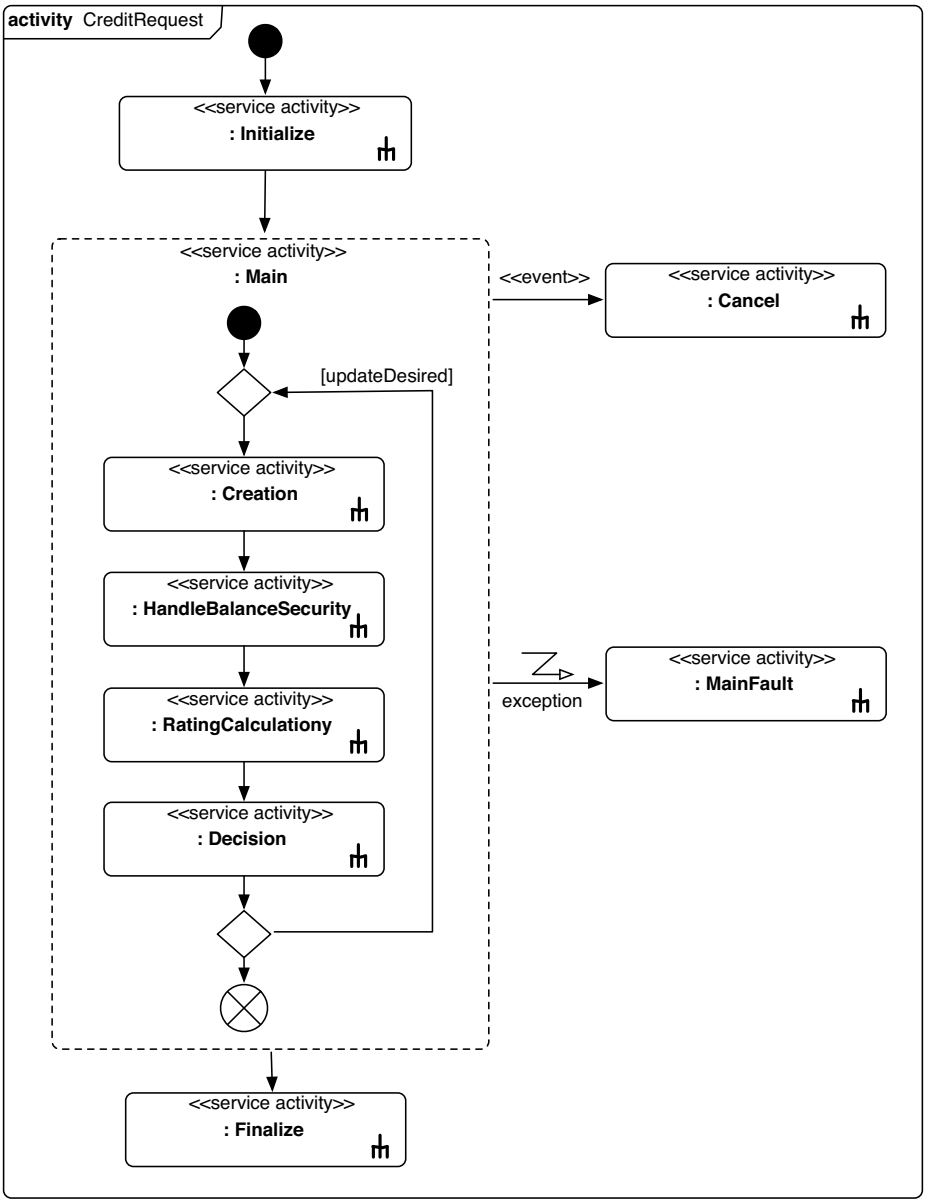


Figure 1: Service Credit Request.

decline, the client is allowed to update the data and restart the request.

At any moment, the client may want to abort the request, in which case the request data needs to be deleted. This requires the execution of compensation activities to semanti-

cally rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests.

3 UML4SOA: A UML Profile for Service Behavior

The *de facto* industrial standard for specifying and documenting software systems, UML, is not expressive enough for modeling structural and behavioral aspects of SOA. SOC introduces key concepts (e.g. partner services, message passing among service requesters and service providers, compensation of long-running transactions) that require specific support in the modeling language to avoid diagrams to become overloaded with technical constructs that reduce their readability.

There are two ways to extend UML to provide a domain-specific high-level design language: One can either change the UML metamodel to create new UML metaclasses or apply a user-defined UML profile to the model. In SENSORIA, the latter approach was chosen, mainly because profiles are easy to use and constitute a lightweight extension. Hence a UML 2.0 profile for modeling the *behavioral* aspects of SOA, called UML4SOA [Fo10], was designed. UML4SOA complements the SoaML profile [OMG], which defines a metamodel for designing the *structural* aspects of SOA. UML4SOA is a minimal extension built on top of the Meta Object Family (MOF) metamodel (i.e. new modeling elements are defined only for specific service-oriented features).

A UML profile consists of a set of stereotypes and constraints (specified in OCL). A stereotype is a limited kind of metaclass that cannot be used by itself, but only in conjunction with one of the metaclasses it extends. A metaclass is a class whose instances are classes. A metaclass defines the behavior of classes and their instances, much like a class in object-oriented programming defines the behavior of certain objects. Each stereotype may extend one or more classes through extensions as part of a profile. All UML modeling elements may be extended by stereotypes (e.g. states, transitions, activities, use cases, components). Stereotypes of the UML4SOA profile can thus be used to enrich UML models with service-oriented concepts, including structural and behavioral aspects of SOA as well as non-functional notions. Constraints allow for a more precise semantics of the newly introduced modeling elements.

The structure of a SOA can be visualized by UML structure diagrams, showing the interplay between components, their ports, and their interfaces. UML4SOA introduces services implemented as ports of components. Each service may contain a required and a provided interface. In turn, interfaces contain one or more operations, which may contain an arbitrary number of parameters. A service has a service description and a service provider, and may have one or more service requesters. These concepts, and the relationships among them, are represented by a metamodel, which provides the basis for the definition of UML4SOA. For each class of the metamodel, a stereotype is defined and relationships are expressed by constraints.

A key aspect of service orientation is the ability to compose existing services, i.e. creating a description of the interaction of several (simpler) services, which is known as

orchestrating. An orchestration is a behavioral specification of a service component, or ≪participant≫ in SoaML. UML4SOA proposes the use of UML activity diagrams for modeling service behavior, in particular for orchestrating (see, e.g., the orchestration in Fig. 1 and the UML4SOA diagrams in Sect. 6). A UML4SOA stereotype ≪serviceActivity≫ can be directly attached as the behavior of a ≪participant≫. The focus is on service interactions, long-running transactions, and their compensation and exception handling.

A scope is a structured activity that groups actions, which may have associated compensation and exception handlers (see, e.g., Fig. 10). Scopes and the corresponding handlers are linked by specific compensation and exception edges (UML4SOA stereotypes ≪compensation≫ and ≪event≫).

Scopes include stereotyped actions like ≪send≫, ≪receive≫, ≪send&receive≫, ≪reply≫, ≪compensate≫, and ≪compensateAll≫. The stereotype ≪send≫ is used to model the sending of messages; ≪receive≫ models the reception of a message blocking until the message is received; ≪send&receive≫ is a shorthand for a sequential order of a send action and a receive action, and ≪reply≫ accepts a return value produced by a previous receive action. Container for data to be send or received are modeled by ≪snd≫ and ≪rcv≫ pins, while ≪lnk≫ pins refer to the partner service in interactions. Long-running transactions, like those provided by services, require the management of compensation. Therefore, UML4SOA contains ≪compensate≫ and ≪compensateAll≫ actions. The former triggers the execution of the compensation defined for a scope or activity, the latter for the current scope. Compensation is called on all subscopes in the reverse order of their completion. The graphical notation of the UML4SOA stereotyped elements is shown in Fig. 2.

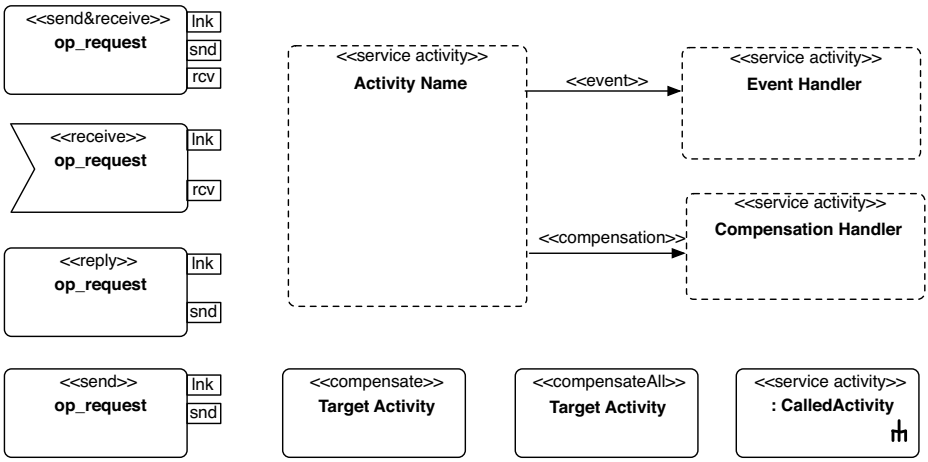


Figure 2: UML4SOA: new graphical elements.

4 Translating UML4SOA Diagrams into UMC Statecharts

Appended to [BM10], we provided a UMC model of the credit request scenario described in Sect. 2. This model was obtained by translating the specification provided by S&N. In the specification discussed in Sect. 2, only the CreditRequest and Rating services are described as UML4SOA activity diagrams, while the other components used by these services (the Portal, Credit Management, Customer Management, Security Analysis, Balance Analysis, and Rating Calculator services) are described by UML4SOA protocol state machines. Since UMC needs a closed model, we moreover modeled a single Client (interacting with a single instance of the Portal) that can make up to two credit requests.

The translation of the main CreditRequest and Rating services was performed in a rather general way, in order to be able to serve as a blueprint for an automatic translation tool from UML4SOA specifications into UMC code (see Sect. 5). This explains the at times long-winded organizational structure. The UMC encoding of the other components, originally specified by protocol state machines, was done by hand.

It is outside the purpose of this paper to try to present all details of the rationale and the rules according to which the UML4SOA activity diagrams were translated into UML statecharts. We just hint here that in general a UML4SOA activity diagram is mapped into a corresponding UML statechart with a single state. Each progression step during the execution of a UML4SOA activity diagram is modeled by one (or sometimes more) transitions in the statechart model, which implement the corresponding semantics. In general, the UML statechart transitions modeling the execution of an activity node have the following form, where tau represents an internal signal of the statechart:

```
s1 -> s1 -- actually no state change
 { tau [enabling conditions for incoming edges] /
       resetting of enabling conditions;
       execution of specific node activity;
       setting of enabling conditions for outgoing edges;
       self.tau;
 }
```

As a specific example, we consider the ≪send≫ node shown in Fig. 3. The corresponding specific UMC transition rule modeling the execution of this node is as follows:

```
s1 -> s1
 { tau [N1_Finalize_DefaultInitialOUT = true] /
       N1_Finalize_DefaultInitialOUT := false;
       LNK_portalService.goodbye([self],
         VAR_CreditRequest_customerData);
       N2_Finalize_Send_goodbyeOUT := true;
       self.tau;
 }
```

Since UML4SOA is a high-level modeling language, some details of the specification

are voluntarily underspecified (oftentimes inherited from UML). This results in the need for specific assumptions to accompany any implementation of a UML4SOA design in a lower-level formalism, so as to make certain details of the model explicit.

UMC offers users the following specific possibilities to customize their system designs:

1. By default, messages that arrive at a time at which they cannot be accepted are never discarded. This specific choice can be reversed by simply avoiding to declare as "Deferred" the corresponding events.

2. By default, the order in which events are removed from an events queue associated with a state machine is "FIFO" (First In First Out). This specific choice can be reversed (even on a *per-object* basis) by simply changing the object queue policy to "RANDOM" (i.e. any queued event is eligible for being dispatched, independently of its position in the queue).

3. By default, all objects are assigned the same priority (thus modeling the maximal degree of nondeterminism in their scheduling). This specific choice can be reversed by simply assigning (also dynamically) to each entity its own priority, thereby modeling more specific schemata.

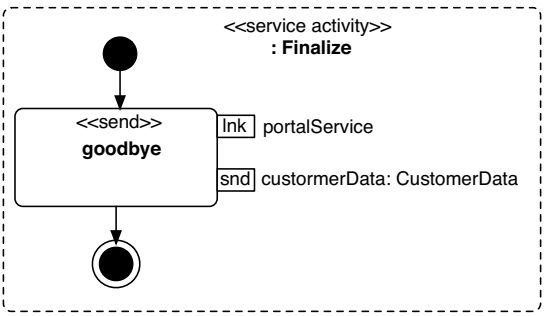


Figure 3: A ≪send≫ node from scope **Finalize**.

5 Tool Support: From MagicDraw to Eclipse to UMC

In this section, we describe the techniques and frameworks we used to support an automatic transformation from UML4SOA designs into UMC executable code. This transformation is based on our translation experience described in the previous section and on OMG's Model Driven Architecture (MDA). MDA is a software development paradigm that focuses the development process more on system design than on its implementation. The main target of this approach is to create models in an efficient and domain-specific way. This is realized by using domain-specific languages and generating the software from these models.

In our context, MDA is used not only to provide a fully automatic approach for transforming UML4SOA service-oriented orchestrations into an implemented system, but also to support qualitative and quantitative verifications by third-party tools. For this purpose we choose the Eclipse Modeling Framework (EMF) and Xpand. EMF is an open source platform that has widely adopted the MDA paradigm and implements most of its core specifications, while Xpand is an EMF-related template language (supported by openArchitectureWare [oAW]) based on domain-specific models for model-to-text transformations. The reason for choosing EMF is threefold:

1. The UML 2.0 core metamodel is already implemented and distributed as part of the Eclipse UML 2.0 plug-in. Hence, there is no need to define a new UML4SOA metamodel, but we only need to consider the stereotypes defined in UML4SOA.

2. MagicDraw can export a UML4SOA design as an Eclipse UML 2.0 instance model.

3. A new graphical interface for UMC, which allows one to directly draw a UMC model in terms of graphical nodes and edges, was already experimented in the context of the Eclipse environment [Su09].

Figure 4 shows the steps performed to obtain UMC code from a UML4SOA system design.

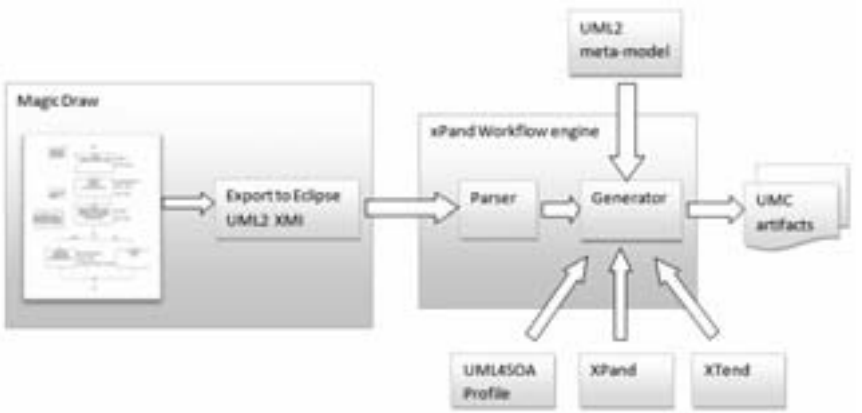


Figure 4: From a UML4SOA system design to UMC executable code.

The Xpand workflow engine executes the transformation process by first invoking a DOM-like parser to create an object graph in memory for our model, and then passing this graph to the Xpand generator in order to get the UMC artifacts. A generator usually consists of one Xpand template file controlling the output generation and a list of Xtend files (another domain-specific language, part of Xpand) containing other user-defined independent functions. An exemplary excerpt from an Xpand template file is given in Fig. 5.

The central concept of Xpand is the template declared using the DEFINE...ENDDEFINE block, which contains the basic structure of the transformation process. A template has a name (UML4SOA2UMC in our example) and is bound to a specific metatype (uml::Model in our example). Inside the main template, a FILE...ENDFILE block creates a new file

Figure 5: Example of an Xpand template.

with the specified name. All code generated inside that block is written to the created output file. Control constructs such as FOREACH and IF are provided as well.

As a specific example of the transformation from UML4SOA to UMC we consider a UML4SOA ≪send≫ node. The corresponding Xpand template code is shown in Fig. 6. This transformation mimicks the manual translation described in the previous section.

Figure 6: Example transformation of a UML4SOA ≪send≫ node.

6 Lessons Learned from the UMC Formalization Experience

A first immediate advantage of the UMC formalization of the credit request scenario from SENSORIA's Finance case study is the possibility to manually explore the graph of all possible system evolutions, verifying through inspection the correctness of the design and the roots of possible anomalies.

In our case, the statespace was not particularly large ($39,530$ states) and even if a small exploration of the initial steps were sufficient to identify some problematic aspects of the design, it is still too large to allow a complete manual inspection.

Model checking SocL formulae proved to be a powerful method for identifying unexpected violations of "supposed-to-be-true" properties, and the detailed explanations provided by UMC (in terms of counterexample/witness paths through the graph of all possible system evolutions) allowed a quick understanding of how such violations might occur. It is outside the purpose of this paper to show the details of the logic (SocL [GM10]) used to analyze our system model of the UML4SOA system design of the credit request scenario. Two exemplary verified properties, here just expressed in plain natural language, are as follows:

1. *"It is always true that if the system accepts a new CreditRequest, then it will either respond to the Client or it will receive a cancel request."*
2. *"If a negative response to a CreditRequest is sent back to the Client, then the rating evaluation was not AAA."*

In the following subsections, we will illustrate in more detail some of the traps and pitfalls that we found in the original system design in UML4SOA as well as inside the UML4SOA definition, and we will show what we believe to be the real underlying issues reflected by them. We will do so abstracting a little from the specific details of the case study, presenting instead small self-contained examples which are not overwhelmed by actually not relevant details.

6.1 Hidden implementation-dependent assumptions inside "high-level" "platform-independent" designs

Suppose we have a Client / Server architecture where the Client performs a login request followed (if successful) by an operation request, while the Server waits for the login request and, if the supplied login data is OK, waits for an operation request to be handled. Using UML4SOA, this simple design can be represented by the diagram shown in Fig. 7 and Fig. 8.

If we adopt a "flat" design for the description of the Server side of this activity (see Fig. 8, left diagram) the semantics is quite clear. However, if we use some additional structured activities inside the Server (see Fig. 8, right diagram) in order to explicitly model the existence of some login and operation mode phases, some semantic subtleties may arise. This is very similar to what happens in our case study, e.g., w.r.t. the **Initialize** and **Main** scopes inside the CreditRequest service.

In the first case (a flat design), once the login response is sent by the Server, the subsequent receive activity can immediately be enabled, while in the second case the two events (completing the send and enabling the receive) are no longer coinciding, since inbetween there is the step of completing the **Login_Phase** and entering the **Operation_Phase**. In principle it might happen that the Server receives an **operation request** before the **Operation_Phase** is entered (and before the receiving of the request is enabled). This makes the

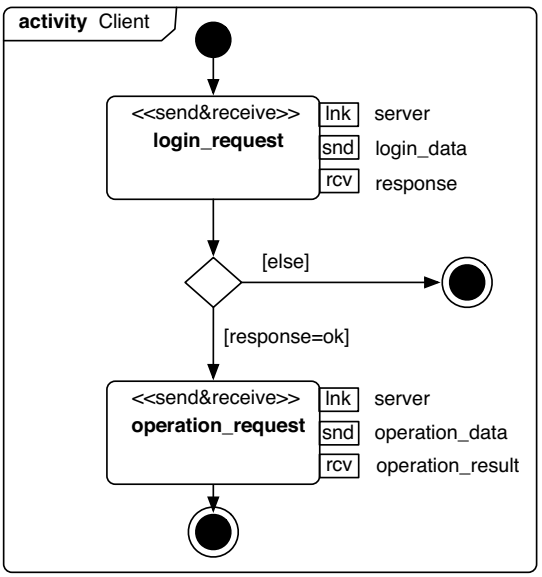


Figure 7: UML4SOA diagrams for the Client.

semantics of the design very implementation dependent as it depends on what is supposed to happen when a message arrives at a system component when the component is not yet ready to accept it (e.g. the request might be queued on the Server until it can be accepted or it might be discarded), and if such a situation should indeed be considered possible.

The problem is that the designer is probably making some *implicit* underlying assumption.

One of these assumptions might be that the time needed by the (remote) Client to receive the login response and produce a new operation request, is much higher than the time needed by the Server to perform all its internal steps from the time at which the response is provided to the time at which the subsequent receive becomes enabled. This kind of assumption might be reasonable, but should be appropriately reflected by the system design, e.g. by stating explicitly that the expected behavior of the Server component is to run as an indivisible activity until it reaches a subsequent communication action.

An alternative assumption, on the contrary, might rely instead on the fact that incoming operation request messages are implicitly supposed not to be discarded, but to be always queued until possibly eventually accepted.

If we do not make any assumption and simply generate a UMC formal model corresponding to the "structured" Server design we can easily detect the presence of deadlocks in the system, e.g. by checking the property that a successful login request is always eventually followed by a response to the operation request. In this way, we can become aware of the existence of some problem in the design. Once the modeling difficulty is understood, we can fix the missing underlying assumptions by explicitly stating one (or both) of the

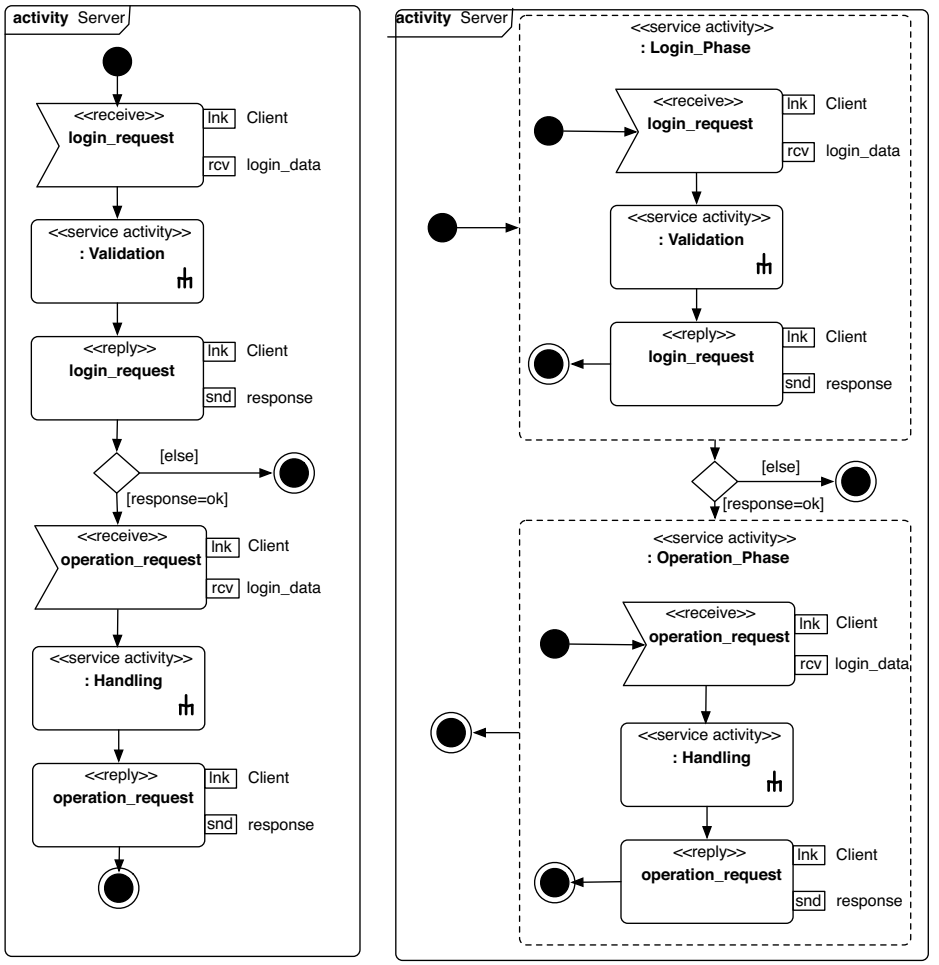


Figure 8: UML4SOA diagrams for the Server (both a flat and a structured design).

possible correct modeling strategies, and these strategies can be efficiently encoded in our UMC model. The case of desired greater granularity of the required Server behavior can be supported in UMC by exploiting the dynamic priorities feature (i.e. raising the priority of a system component for a short period of time in order to execute a sequence of internal steps as a unique indivisible activity). The case of incoming messages not supposed to be discarded even if arriving at a time in which a component is not yet able to handle them, on the other hand, is supported by UMC by allowing to define the corresponding events as "deferred" (in the UML statecharts sense) to achieve the behavior of queued messages. Once any (or both) of these solutions is adopted we can verify that the system behavior becomes again, as intuitively expected, equivalent to that of the flat model of the Server.

These problems can be exacerbated by a third kind of hidden assumption, shown in Fig. 9.

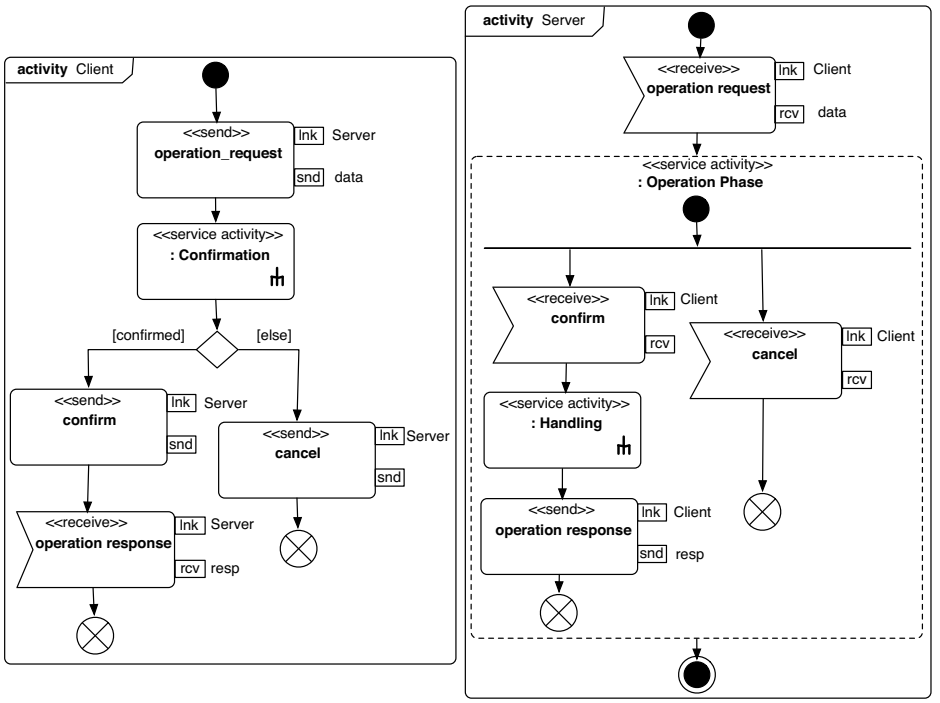


Figure 9: UML4SOA diagrams for the Client and the Server — second variant.

In the variant depicted in Fig. 9, the Client is supposed to ask the Server for a certain operation, with the possibility to later cancel or confirm this operation. Similarly, the Server is initially supposed to wait for an operation request, after which an event **Operation Phase** is entered and, depending on the Client's decision, the appropriate activity is performed. The design of Fig. 9 seems correct at a first glance, but what happens if the **operation request** message is delayed by the network and happens to be delivered after the **confirm** or **cancel** message? Again, this is a strictly implementation-dependent situation (due to UML allowing so-called semantic variation points) and if the implementation is based on the assumption that wrongly timed messages are discarded, then the system deadlocks. Also in this case we have an apparently correct "platform-independent" "high-level" design which, on the contrary, just hides specific implementation-dependent assumptions.

When building a UMC model, the appropriate choice of modeling a communication network that can or cannot deliver the messages in a different order from the one in which they are sent, can be easily made by specifying a specific queueing policy (RANDOM versus FIFO) for the events arriving to the system components.

6.2 Uncertain semantics of UM4SOA features

Informally describing a language feature is one thing, formally reasoning on all the possible consequences of a language choice is quite another thing. This could be the summary of this subsection. This time the sample culprit is the design of the compensation mechanism of UML4SOA.

UML4SOA allows one to associate to a service activity region a ≪compensation≫ edge leading to another compensation service activity, to be executed whenever the effects of a successful completion of the original service activity have to be undone in some way. This compensation activity is requested by the execution of a ≪compensate≫ or ≪compensateAll≫ action. The main problem here is that compensation handling is a concept that is tightly coupled to that of an "atomic transaction" (i.e. an activity with an "all or nothing" semantics), but UML4SOA service activities are not required to have such a transactional semantics.

In the credit request scenario, there are several cases in which non-transactional activities are associated to compensation handlers, with the consequence that some expected system property do not hold. This typical example is shown (in a form equivalent to what happens e.g. inside the **Creation** activity of the case study) in Fig. 10.

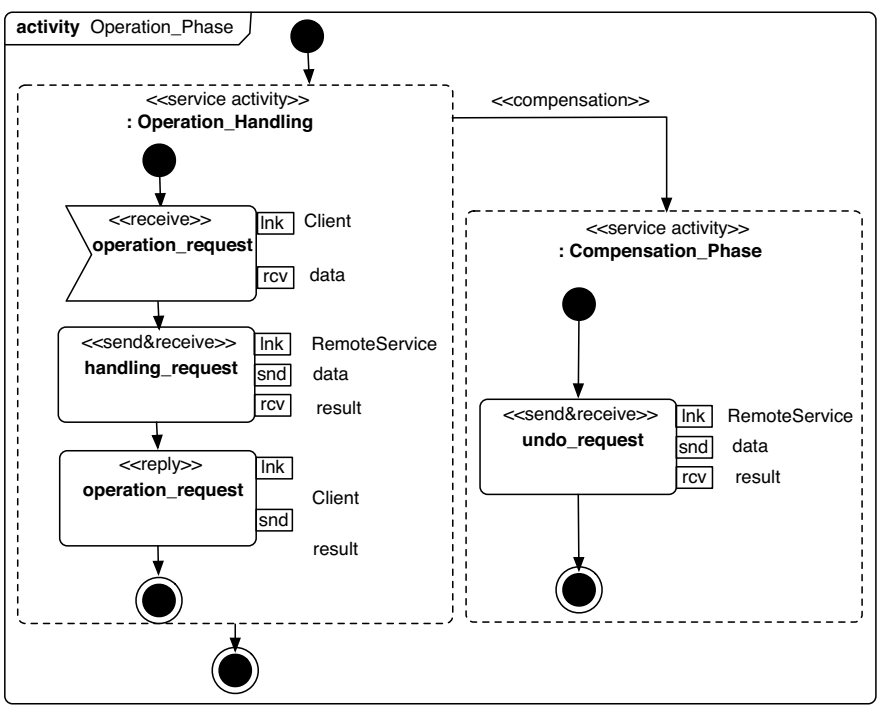


Figure 10: UML4SOA compensation.

In Fig. 10, the compensation activity consists of requesting, with the **undo_request** ser-

vice call, the "undo" of the **handling_request** service call executed inside the **Operation_Handling** activity. This logically means that "orphan" (i.e. not associated to successful credit requests) **handling_request** instances should not be present in the system. However, if a compensation request is executed after the **handling_request** service call is issued but before the **Operation_Handling** activity is completed, then no compensation is activated and an orphan **handling_request** instance still exists.

In this case, at least two kind of problems are raised and evinced by the formal modeling:

1. The semantics of compensation should probably be tied with an atomic transaction mechanism, a fact that is not sufficiently well described by the current UML4SOA definition of service activities and compensations.

2. The designer of this fragment of activity might have implicitly assumed that the execution of the **Operation_Handling** activity cannot be interrupted from the outside. Since it contains no error-path, it is supposed to always complete successfully. Unfortunately, the overall system design allows two parallel threads to asynchronously interfere with one another, and at first the impact of this possibility was not well understood in the case study. In the credit request scenario, the asynchronous interfering activity is the one originated by a **cancel** operation triggered by the Client.

Another subtle and potentially problematic aspect of the UML4SOA semantics of compensation, is related to the order in which the subactivities of a given activity have to be compensated. While UML4SOA requires that subactivities need to compensated in precisely the reverse order w.r.t. their completion order, this is not exactly what is required by the BPEL semantics (which is more lazy and allows violation of completion ordering for not causally related subactivities). In our case study, this difference is absolutely irrelevant, but in principle more complex designs could be imagined in which the difference does become observable.

If we suppose that software development approach relies on an automatic translation from UML4SOA into BPEL, directly mapping UML4SOA compensations into BPEL compensations, then for more complex designs it might happen that an apparently working system from a certain point onwards starts to show unexpected anomalies. This might happen, e.g., when the adopted BPEL execution engine passes from one version to another, or from one vendor to another. In this case, formal methods might be of help by rigorously defining the assumptions and the meaning of the high-level designs. However, they can do little w.r.t. to the verification of compatibility issues of BPEL execution engines, and even less w.r.t. the inconsistencies raised by the evolutions in time of the BPEL specification itself.

During the formalization of the credit request scenario from SENSORIA's Finance case study, several other aspects have raised discussions and clarification requests among the different partners involved in the project. The semantics of UML4SOA protocol state machines that are used to model the unspecified components of the system, e.g., was found not well specified, nor was the semantics of service instance creation and service connection establishment. For all these aspects, the formal modeling effort offered a good opportunity to clarify and disambiguate the intended meaning of the language features, a

process which is probably still not complete.

6.3 Hidden complexity of scarcely structured designs

For several decades now, the danger of using "goto" as a control flow command has been widely recognized in the field of software engineering. This is due to the difficulty of analysis and understanding that its use introduces in the algorithms. It is therefore hard to understand why nowadays, when designing "high-level" "platform-independent" design languages, we resort to recycling the goto construct just because with a graphic design notation, nodes and edges are the easiest graphical elements to design. The result can quickly become a "spaghetti design" (as shown in the leftmost diagram of Fig. 11), with the consequence that the true behavior of the system becomes obscure and difficult to understand in all its ramifications.

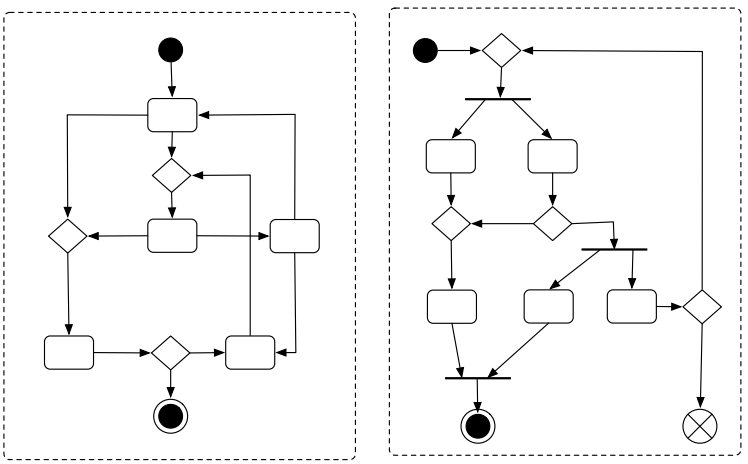


Figure 11: Weren't gotos considered harmful?

Moreover, in almost all programming languages great care has been given to the introduction of concurrent features (task, threads, co-routines) with the design goal of well identifying the concurrently evolving activities and, above all, keeping the flow of concurrent elements independent as much as possible. We repeat that it is therefore hard to understand that "high-level" design languages resort to the use of low-level graphical elements, like "fork" and "join", to handle concurrency, thus potentially allowing an even nastier form of spaghetti design (as shown in the rightmost diagram of Fig. 11). The situation is even made potentially more dangerous by exploiting elements like "activity final" nodes (killing all concurrent subactivities inside a parallel activity) or by raising exceptions, which might have the consequence that an error in one concurrent subactivity asynchronously terminates other concurrent activities in an unclear, implicit, or uncontrolled way.

Fortunately, in our case study we did not have to face such an abuse of unstructured constructs. However, we cannot avoid to remark that one of the design flaws present in the

service orchestration (as shown in the previous subsection) was in fact caused by a bad interference between two concurrent activities. The overall situation actually occurring in our case study is like that shown in Fig. 12, in which the **Operation_Phase** is unexpectedly aborted because of an exception raised inside the **Cancel** activity, asynchronously triggered as a concurrent flow by an external event.

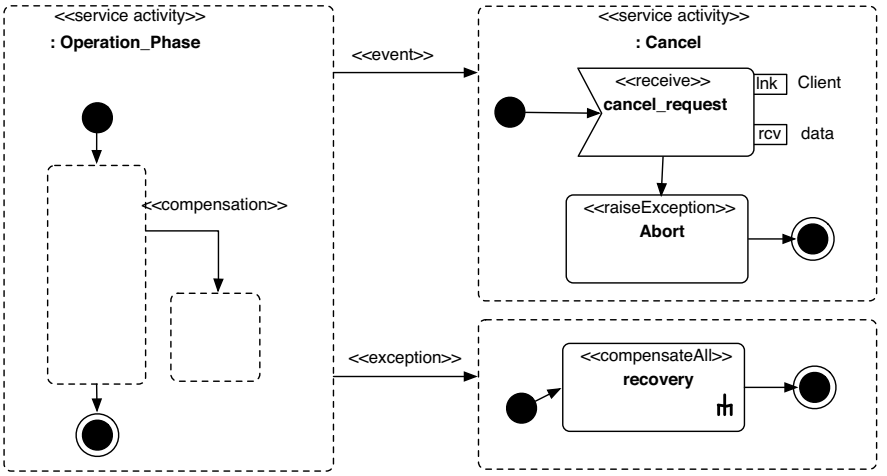


Figure 12: UML4SOA compensation — second variant.

7 Conclusions

High-level graphical design notations are very intuitive and very efficient in describing the current structure and status of an ongoing software project. Especially if they are associated with automatic code generation / design verification features, they might play an interesting role inside an agile software development process as they can help in reducing the effort of evolution cycles, and they can help in maintaining the focus of the development at a level which is also understandable by the client. This might facilitate the cooperation between the clients and the developers, promoting the rapid delivery of software and its regular adaption to the evolving requirements, in the spirit of the agile approach to software development.

In this paper we show, however, that they may also be a source of problems, most of which the use of formal methods can avoid. We illustrate our claim by generalizing examples that we encountered in a case study. The three types of problems we focus on are hidden implementation-dependent assumptions inside “high-level” “platform-independent” designs, uncertain semantics of features of “high-level” design languages, and hidden complexity of scarcely structured designs.

It is our firm belief that high-level graphical design notations always need to be backed up by a precise and rigorous semantics, i.e. by formal methods, especially when they are

planned to be used inside agile software processes. The adoption of a rigorous or formal semantics for these notations, and the adoption of formal verification methods allow to explore and understand in all their hidden ramifications the high level designs. Automatic formal model generation from high level graphical designs is not only desirable but also pragmatically feasible e.g. using advanced model transformation techniques, and this is particularly valuable in the context of agile development approaches which are supposed to exploit the rapid and continuous updates of the system under development.

Acknowledgements

We thank all our partners in SENSORIA for detailed discussions of the Finance case study, but in particular Jannis Elgner from S&N, Philip Mayer and Martin Wirsing from LMU, Lucia Acciai, Federico Banti, Francesco Tiezzi and Rosario Pugliese from DSIUF, and Stefania Gnesi from ISTI.

References

[Be10] M.H. ter Beek, A. Fantechi, S. Gnesi and F. Mazzanti, A state/event-based model-checking approach for the analysis of abstract system properties. To appear in *Science of Computer Programming*, 2010.

[BM10] M.H. ter Beek and F. Mazzanti, Modelling and Analysing the Finance Case Study in UMC. Technical Report 2010-TR-007, ISTI–CNR, 2010.

[FH01] M. Fowler andJ. Highsmith, The Agile Manifesto. Software Development, Augist 2001. (see also http://www.agilemanifesto.org)

[Fo10] H. Foster, L. Gönczy, N. Koch, P. Mayer, C. Montangero and D. Varró, UML Extensions for Service-Oriented Systems. In [WH10], 2010.

[GM10] S. Gnesi and F. Mazzanti, An Abstract, on the Fly Framework for the Verification of Service Oriented Systems. In [WH10], 2010.

[Ma09] F. Mazzanti, Designing UML models with UMC. Technical Report 2009-TR-43, ISTI–CNR, 2009.

[Mar02] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR Upper Saddle River, NJ USA, 2002

[OMG] Object Management Group, Service oriented architecture modelling Language (SoaML): Specification for the UML Profile and Metamodel for Services (UPMS), April 2009. `http://www.omg.org/cgi-bin/doc?ptc/09-04-01/`

[oAW] openArchitectureWare. `http://www.eclipse.org/workinggroups/oaw/`

[Pa07] M.P. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer* 40, 11 (2007), 38–45.

[SH05] M.P. Singh and M.N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005.

[Su09] A. Sulova, Model Driven Software Development con Eclipse, StatechartUMC. In *Proceedings of the 4th Italian workshop on Eclipse technologies (Eclipse-IT 2009), Bergamo, Italy* (A. Gargantini, Ed.), Eclipse Italian Community, 2009, 113–114. An extended version appeared as Technical Report 2009-TR-050, ISTI–CNR, 2009. In Italian.

[S&N] S&N AG. `http://www.s-und-n.de/`

[SW07] J. Shore and S. Warden, The art of agile development. OReilly, 2007.

[UMC] UML Model Checker. `http://fmt.isti.cnr.it/umc/`

[UML] UML4SOA. `http://www.uml4soa.eu/`

[WH10] M. Wirsing and M. Hölzl (Eds.), *Rigorous Software Engineering for Service-Oriented Systems: Results of the* SENSORIA *project on Software Engineering for Service-Oriented Computing*, Springer, 2010. To appear. See also `http://www.sensoria-ist.eu/`

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Language to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Busi-ness 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Erfahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT: Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik – Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwick-lung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^2CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.)
10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on Electronic Voting 2010
co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2010)
Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.)
German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.)
perspeGKtive 2010
Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für die Informatik
Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science – Proceedings of ISSS and BPSC

P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM`2010
Second International Workshop on Formal Methods and Agile Methods

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de