

# Seamless Method- and Model-based Software and Systems Engineering

Manfred Broy

Institut für Informatik, Technische Universität München  
D-80290 München Germany, broy@in.tum.de  
<http://wwwbroy.informatik.tu-muenchen.de>

**Abstract.** Today engineering software intensive systems is still more or less handicraft or at most at the level of manufacturing. Many steps are done ad-hoc and not in a fully systematic way. Applied methods, if any, are not scientifically justified, not justified by empirical data and as a result carrying out large software projects still is an adventure. However, there is no reason why the development of software intensive systems cannot be done in the future with the same precision and scientific rigor as in established engineering disciplines. To do that, however, a number of scientific and engineering challenges have to be mastered. The first one aims at a deep understanding of the essentials of carrying out such projects, which includes appropriate models and effective management methods. What is needed is a portfolio of models and methods coming together with a comprehensive support by tools as well as deep insights into the obstacles of developing software intensive systems and a portfolio of established and proven techniques and methods with clear profiles and rules that indicate when which method is ready for application. In the following we argue that there is scientific evidence and enough research results so far to be confident that solid engineering of software intensive systems can be achieved in the future. However, yet quite a number of scientific research problems have to be solved.

**Keywords:** Formal methods, model based development

## 1 Motivation

Since more than four decades, extensive research in the foundations of software engineering accomplished remarkable results and a rich body of knowledge. Nevertheless the transfer to practice is slow – lagging behind the state of science and sometimes even not making much progress.

In the following we direct our considerations both to technical aspects of development and to management issues. In fact, a lot of the problems in software and systems engineering do not come from the software engineering techniques but rather from problems in project management as pointed out by Fred Brooks (see [2]). His book, “The Mythical Man-Month”, reflects experience in managing the development of OS/360 in 1964-65. His central arguments are that large projects suffer from management problems different in kind than small ones, due to division in labour, and

their critical need is the preservation of the conceptual integrity of the product itself. His central conclusions are that conceptual integrity can be achieved throughout by chief architects and implementation is achieved through well-managed effort.

The key question is what the decisive success factors for software and systems engineering technologies are. What we observe today is an enormous increase in functionality, size and complexity of software-based systems. Hence, one major goal in development is a reduction of complexity and a scaling up of methods to the size of the software. Moreover, technologies have to be cost effective. Techniques that contribute to the quality of the product but do not prove to be affordable are not helpful in practise.

## 2 Engineering Software Intensive Systems

Engineering is the systematic application of scientific principles and methods to the efficient and effective construction of useful structures and machines. Engineering of software intensive systems is still a challenge! The ultimate goal is to improve our abilities to manage the development and evolution of software intensive systems. Primary goals of engineering software intensive systems are suitable quality, low evolution costs and timely delivery.

### 2.1 Engineering and Modelling based on First Principles

To make sure that methods are helpful in engineering and really do address key issues, it is advisable to base development methods on principles and strategic goals. These principles are valid conclusions of the experiences gained in engineering software intensive systems. In the following we recapitulate a number of such principles and then discuss to what extent these principles can be backed up by specific methods.

One of the simple insights in software and systems engineering is, of course, that not only the way artefacts are described and also not just the methods that are applied are most significant for the quality of the development outcome. What is needed is a deep understanding of the engineering issues taking into account all kinds of often not explicitly stated quality requirements. As engineers, we are interested to be sure that our systems address the users' needs in a valid way and show required quality, for instance, that they are safe with a high probability and that during their lifecycle they can be evolved and adapted to the requirements in the future to come. In particular, evolving legacy software is one of the nightmares of software engineering. One major goal is keeping software evolvable. It is unclear to what extent available methods can help here.

The discipline of systems and software engineering has gathered a large amount of development principles and rules of best practice. Examples are principles like:

- separation of concerns
- stepwise refinement
- modularity and compositionality

- hierarchical decomposition
- standardized architecture and patterns
- abstraction
- information hiding
- rigor and formality
- generality – avoiding overspecification
- mitigation of risk
- incremental development
- anticipation of change
- scalability

Software Engineering “Maxims” say:

- Adding developers to a project will likely result in further delays and accumulated costs.
- Basic tension of software engineering is in trade-offs such as:
  - Better, cheaper, faster — pick any two!
  - Functionality, scalability, performance — pick any two!
- The longer a fault exists in software
  - the more costly it is to detect and correct,
  - the less likely it is to be properly corrected.
- Up to 70% of all faults detected in large-scale software projects are introduced in requirements and design.
- Insufficient communication and transparency in the development team will lead to project failure.
- Detecting the causes of faults early may reduce their resulting costs by a factor of 100 or more.

How can specific development methods evaluate these rules of thumb and support these principles? Some only address management issues.

## **2.2 From Principles to Methods, from Methods to Processes**

Given principles, we may ask how to derive from known and proven methods and from methods development processes.

### **2.2.1 Key Steps in Software and Systems Engineering**

In fact, looking at projects in practise we may identify the key activities in software and systems engineering. When studying projects and their success factors on the technical side, the latter prove to be always the same, namely, valid requirements, well worked out architectures both addressing the needs of the users as well as the application domain and an appropriate mapping onto technical solutions and adequate and proactive management. Constructive and analytical quality assurance is essential. However, only a part of it is verification not to forget validation of the requirements. An important issue that is not addressed enough in methods and techniques is comprehensibility and understandability. Formal description techniques are precise,

of course. But the significant goal is reduction of complexity and ease of understanding – this is why graphical description techniques seem to be so popular! Engineers, users, stakeholders have to understand the artefacts worked out during the development process. Often understanding is even more important than formality. Up to 60 % and more of the effort spent in software evolution and maintenance is code understanding. If a formal method precisely captures important properties, but if engineers are not able to understand it properly the way it is formulated then the method is not useful in practice.

### 2.2.2 Requirements Engineering

Gathering requirements based on collecting, structuring, formalizing, specifying, modelling are key activities. One of the big issues is, first of all, capturing and structuring valid requirements. IEEE standard 830-1998 mentions the following quality attributes of requirements documentation. Requirements specification and documentation has to be correct, unambiguous, complete, consistent, ranked for importance/stability, verifiable, modifiable, and traceable.

For effective requirements engineering we do not necessarily need formality of methods to begin with, since from the attributes listed above mainly consistency, unambiguity, and verifiability are supported directly by formal methods. A much better systematics for requirements is needed.

### 2.2.3 Architecture Design

Architecture aims at structuring software systems. Usually there is not just one architectural view onto a software system, but many related architectural viewpoints. A comprehensive architecture includes the most significant viewpoints. We give a short overview over viewpoints as they proved to be useful in automotive software and systems engineering:

- *Usage Process Viewpoint*: user processes and use cases,
- *Functional Viewpoint*: decomposition of systems into system function hierarchy, dependencies, functional specification of functions and their dependencies,
- *Logical component viewpoint*: decomposition of systems into data flow networks of logical system components, component hierarchy, component interface specification,
- *Technical viewpoint*: realization of logical components by software modules, run time objects, deployment and scheduling of run time objects onto the software and hardware platform.

For the design of architecture at a logical level we suggest hierarchies of logical components, on which a detailed design can be based. The difference between architecture and detailed design [13] is expressed as follows:

- Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design.

- Design is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements.

A detailed design is mandatory for systematic module implementation and verification and later system integration, which is the key step in system development, including integration verification.

## **2.3 Empirical and Experimental Evaluation of Software Engineering Principles and Methods**

We have listed a number of principles that are widely accepted as helpful and valid in the development of large software systems. In addition to these principles we find suggestions such as agile development (see scrum). Another example of recent claims for improvement are so-called service-oriented architectures where it is claimed that these architectures lead to a much better structuring in particular, of business and web-based systems. A third example, which is more on the programming language side is aspect-oriented programming that claims to lead to better structured programs which are easier to change and to maintain. A scientific justification or disproof of such claims is overdue.

### **2.3.1 Justifying Claims about Principles and Methods**

One idea is that we can use empirical results and experiments to prove that certain methods and principles are effective. However, this idea runs into severe difficulties. One difficulties is that still the field of software and systems engineering is changing too fast such that experiences made a few years ago may not apply for the next system because our hardware changes, our software techniques change, the size of the system changes.

The other problem is the fuzziness of statistical analysis. There are too many factors, which influence the success of software projects. If we apply a method in one project and not in the other and then compare both projects it is hardly possible to conclude from the success of the first project the effectiveness of the method. There are so many additional project influences such that it is hard to determine which one is dominantly responsible for project success.

A typical example is the application of model-based technology. When doing so the idea is that we model the requirements and architecture. If we then conclude that there is an improvement in the effectiveness of software development by using model-based techniques, the true reason might be that it is not so much the application of modelling techniques but rather the systematic way of dealing with requirements and architecture.

Of course, we have clear indications that firm systematic development approaches improve the quality of software but, of course, we always have to ask about the costs of the development taking to account that software development is very costly. It is not possible just to look at the methods and their effects on development quality. We

have also to determine how cost-effective a method is and then relate the positive effects of the methods to its costs as addressed under keywords as “design to cost”.

### **3 Scientific Foundations of Engineering Methods**

First of all, formalization is a general method in science. It has been created as a technique in mathematics and also in philosophical and mathematical logic with the general aim to express propositions and to argue about them in a fully objective way. In some sense it is the ultimate goal of science to deal with its themes in an absolutely objective way.

Only in the last century, formal logic has entered into engineering. First of all, logic has been turned into an engineering concept when designing switching circuits and also by the logic of software systems. Secondly, the logical approaches help in developing software and digital hardware – after all code is a formal artefact defining a logic of execution.

#### **3.1 About the Concept of Engineering Methods**

A method defines “how to do or make something”.

A method is a very general term and has a flavour that it is a way to reach a particular goal, where the steps to reach that goal are very well defined such that skilled people can perform them. Engineers therefore heavily use methods as ways to reach their sub-goals in the development process.

#### **3.2 Why Formal Specification and Verification is Not Enough**

Formal development methods that mainly aim at formal specification and verification are not sufficient for the challenge to make software systems reliable and functionally safe such that they fulfil valid requirements of their users with expected quality and are constructed in cost effective ways. Pure formalization and verification can only prove a correct relationship between formal specifications and implementations but cannot prove that the systems meet valid requirements.

Therefore the project on the verifying compiler (see [11]) has an essential weakness since it only addresses partial aspects of correctness but not validity of requirements.

#### **3.3 Importance of the Formalization of Engineering Concepts**

Engineering concepts in systems and software development are mostly complex and abstract. Therefore they are difficult to define properly, to understand and justify. We see a great potential for formalization in the precise definition of terms and notions in engineering and in the formal analysis of engineering techniques. We see a significant

discrepancy between a formal method and the scientific method of formalization and formal foundations of engineering method.

### 3.4 The Role of Automation and Tools

Any methods used in the engineering of software systems are only helpful if they scale up and are cost effective. This means they have to be supported to a great deal by automation through tools.

Here formal methods actually can offer something because any tool support requires a sufficient amount of formalization. The better a method can be formalized the better it can be automatized and supported by tools.

## 4 Seamless Model Based Development

*Model Based Engineering (MBE)* is a software development methodology which focuses on creating models, or abstractions, more close to particular domain concepts rather than programming, computing and algorithmic concepts. It aims at increasing productivity by maximizing compatibility between systems, simplifying the process of design, increasing automation, and promoting communication between individuals and teams working on the system.

### 4.1 What are Helpful Models?

A model is an *appropriate abstraction for a particular purpose*. This is of course, a very general connotation of the concept of a model. Having a closer look, models have to be represented and communicated in order to be useful in software engineering. We should keep in mind, however, that an appropriate “Gedankenmodell”, which provides a particular way and abstraction of how to think about a problem is useful for the engineer even without an explicit representation of models for communication. Using Gedankenmodells means to think and argue about a system in a goal directed way in terms of useful models.

We are interested not only in individual models but also in *modelling concepts*. These are hopefully proven techniques to derive certain abstractions for specific purposes. Here is an important strength of modelling, namely that effective modelling concepts provide useful patterns of engineering such as design patterns.

#### 4.1.1 Modelling Requirements

Capturing and documenting requirements is one of the big challenges in the evolution of software intensive systems. As well known, we have to distinguish between functional requirements and quality requirements. We concentrate in the following mainly on functional requirements. For functional requirements modelling techniques help since we can describe the functionality of software-intensive systems by using formal specification techniques.

Systematic requirements engineering produces complete formal specifications of the interface behaviour of the system under construction. Since for many systems the functionality is much too large to be captured in one monolithic specification, specifications have to be structured. For instance, techniques are needed to structure the functionality of large multifunctional systems by hierarchies of sub-functions. The system model, briefly introduced in the appendix of [9], allows specifying the interface behaviour of the sub-functions and at the same moment using modes to specify how they are dependent and to capture the feature interactions. Worked-out in full detail state-machines with input and output capture the behaviour of the sub-services describing the interactions and dependencies between the different sub-functionalities with the help of modes. Such descriptions are worked-out starting from use-cases.

Thus way we obtain a fully formalized high level functional specification of a system structured into a number of sub-functions.

#### **4.1.2 Modelling Architecture**

A key task in system evolution is the modelling of architectures. Having modelled the function hierarchy as described above a next step is to design a logical component architecture capturing the decomposition of the system into logical components, again in a hierarchical style. However, logical component architectures provide completely different views in contrast to function hierarchies derived in requirements engineering.

How are the two views related? The interface behaviour of the logical architecture hierarchy has to be correct with respect to the overall functionality as specified by the function.

#### **4.1.3 From Requirements and Architecture to Implementation and Integration**

Having worked out a complete description of requirements and architecture all further development steps are very much guided by these artefacts. First of all, the architecture model provides specifications of the components and modules. On this basis, we can do a completely independent implementation of the individual components (following the principle of separation of concerns and modularity), say, in terms of state machine models. From these state machine models we can generate code. Moreover, we can even formally verify architectures before having worked out their implementation. Architectures with components described by state machines can be tested even before implementation and from them we can generate test cases for integration tests. Provided, the architecture is described and specified in detail, we can derive and verify from the architectural specification also properties of the overall functionality as specified by the function hierarchy specification of the system.

Architecture design can be carried out rigorously formally. This is, of course, not so easy for large systems. It is a notable property of formal methods whether they scale and how they may be applied in a lightweight manner.

If early architecture verification is done accurately and if modules are verified properly then during system integration we have not to be afraid of finding many new bugs. In fact, if architecture verification and component verification are not done properly, significant bugs are discovered much too late during system integration, as it is



the case in practice today, where architectures are not verified and modules are not properly specified. Then module verification cannot be done properly; all problems of systems show up only much too late during system integration and verification.

## 4.2 Modelling Systems

Based on a comprehensive set of concepts for modelling systems – as shortly outlined in the appendix of [9] – an integrated system description approach can be obtained.

### 4.2.1 The Significance of Precise Terminology and Clean System Concepts

One of the significant advantages of formal and mathematical techniques in software and systems engineering is not just the possibility to increase automatic tool support, to formalize and to write formal specifications and to do formal verification. Perhaps, equally important is to have clear notions and precise terminology. In many areas of software and systems engineering terms are complex abstract notions, are fuzzy and not properly chosen and made precise. Simple examples are terms like “function” or “feature” or “service”, which are frequently used in software and systems engineering without a proper definition. As a result understanding between the engineers is limited and a lot of time is wasted in confusing discussions.

### 4.2.2 An Integrated Model for System Specification and Implementation

A specified and implemented system is described by (for models of the used formal concepts see appendix of [9]):

- an identifier  $k$ , the system name,
- an interface behaviour specification consisting of
  - a syntactic interface description  $synif(k)$
  - an interface behaviour specification  $specif(k)$
- an implementation design  $dsgn(k)$  for the interface syntactic interface, being either
  - an interpreted architecture  $dsgn(k)$ ,
  - a state machine  $dsgn(k)$ .

We end up with a hierarchical system model that way, where systems are decomposed into architectures with subsystems called their *components* that again can be decomposed via architectures into subsystems until these are finally realized by state machines. We assume that all identifiers in the hierarchy are unique. Then a hierarchical system with name  $k$  defines a set of subsystems  $subs(k)$ .

Each subsystem as part of a specified and implemented system then has its own specification and implementation. A system has an *implemented behaviour* by considering only the implementation designs in the hierarchy and a *specified behaviour* by the interface specifications included in the hierarchy.

A system  $k$  is called *correct*, if the interface abstraction of its implementation  $A = dsgn(k)$  has an interface abstraction  $F_A$  that is a refinement of its interface specification  $specif(k) = F$ :

$$F \approx_{\text{ref}} F_A$$

On the basis of such a formal system model we can classify faults. A system is called *fully correct*, if all its sub-systems are correct. A system is called *faulty*, if some of its subsystems are not correct. A system fault of a system implemented by some architecture is called *architecture fault*, if the interface behaviour of the specified architecture is not a refinement of the interface specification of the system. A fault is called *component fault*, if the implemented behaviour of a subsystem is not a refinement of the specified behaviour. A clear distinction between architecture faults and component faults is not possible, in practice, today due to insufficient architecture specification (see [15]).

### 4.2.3 From Models to Code

If, as part of the architectural description of a system, for each component an executable model in terms of state-machines is given then we have an executable system description. With such a description we can generate test cases both at the component level, at the integration level and at the system test level. Then if the code is handwritten it can be tested by the test cases generated from the system model. On the other hand it is also possible to generate the code directly from the complete system description. In this case it does not make much sense to generate test cases from the models since the generated code should exactly reflect the behaviour of the models. Only if there is some doubt whether the code generator is correct to makes sense to use test cases to certify the code generator.

### 4.2.4 Software product lines

It is more and more typical that software development is no longer done from scratch where a completely new software system is developed in a green field approach. It is much more typical that systems are developed in the following constellations:

- A system is to be developed to replace an existing one where parts of the older systems will be reused.
- Significant parts of a system are taken from standard implementations and standard components available.
- A system evolution is carried out where a system is step by step changed and refined in adapted to new situations.
- A software family has to be produced where typically a large number of systems, with very similar functionalities, have to be created, in such cases a common basis is to be used.
- A platform is created which allows implementing a family of systems with similarities.

Often several of these development patterns are applied side by side. Model-based technologies are very well suited to support these different types of more industrialized software development.

#### 4.2.5 Modular System Design, Specification, and Implementation

It is essential to distinguish between

- the architectural design of a system and
- the implementation of the components specified by an architectural design.

An architectural design consists in the identification of its components, their specification and the way they interact and form the architecture.

If the architectural design and the specification of the components is precise enough then we are able to determine the result of the cooperation of the components of the architectures, according to their specification, even without providing an implementation. If the specifications are addressing behaviour of the components and if the design is modular, then the behaviour of the architecture can be derived from the behaviour of its components and the way they are connected. In other words, in this case architecture has a specified behaviour. This specified behaviour has to be put into relation with the specification of the requirements for the system.

Having this in mind, we obtain two possibilities in making use of architecture descriptions. First of all, architecture verification can be done, based on the architecture specification without having to give implementations for the components. How verification is done depends on how the components are described. If component specifications are given by abstract state machines, then the architecture can be simulated and model-checked (if it is not too big). If component specifications are given by descriptive specifications in predicate logic, then verification is possible by logical deduction. If the components are described informally only, then we can design test cases for the architecture to see whether architectures conform to system specifications.

Given interface specifications for the components we can first of all implement the components, having the specifications in mind and then verify the components with respect to their specifications. So, we have two levels of verifications, namely, *component verification* and *architecture verification*. If both verifications are done carefully enough and if the theory is modular then correctness of the system follows from both verification steps as a corollary.

Finally, if for an implemented system for a specified system and we distinguish faults in the architectural design, where we may identify in which stage which faults appear the architecture verification would fail, from faults in the component implementation. Note that only if we are careful enough with our specification techniques to be able to specify architectures independent from component implementations then the separation of component test, architecture and integration tests and system tests are meaningful.

Furthermore, for hierarchical systems the scheme of specification, design, and implementation can be iterated for each sub-hierarchy in the architecture. In any case, we may go on in an idealised top-down development as follows: We give a requirements specification for the system, we carry out an architectural design and architectural specification for the system, this delivers specifications for components and we can go on with component specifications as requirements specification for the successive step of designing and implementing the components.

#### 4.2.6 Formal Foundation of Methods and Models

As defined, a formal method (or better a formal engineering) method applies formal techniques in engineering. Another way to make use of formalization is the justification of methods by formal theories. Examples are proofs that specification concepts are modular or that concepts of refinement are transitive or that transformation and refactoring rules are correct.

Formal justification of methods or modelling techniques is therefore important. This allows justifying methods or development rules to be used by engineers without further explicit formal reasoning.

### 5 Seamless Modelling

Being formal is only one attribute of a modelling technique or a development method. There are others – not less important.

#### 5.1 Integration of Modelling Techniques

Modelling techniques and formal techniques have one thing in common. If they are applied only in isolated steps of the development process they will not show their full power and advantages well enough and, as a result, they often will not be cost effective. If just one step in the development process is formalized and formally verified and if formally verified programs are then given to a compiler, which is not verified, it is unclear whether the effect of the formal verification brings enough benefits.

The same applies to modelling. When high level models of systems are constructed and a number of results have been achieved based on these models, it is not cost effective if then in the next step the model is not used anymore and instead the work is continued by working out different models and solutions.

Tab. 1 shows a collection of modelling and development methods as well as their integration into a workflow aiming at a seamless development process by formal models and formal methods. In requirements engineering the first result should be the function hierarchy as described above. Then the logical component architectures are designed and verified by test cases generated from the specification of the functional hierarchies. For the components, test cases can be generated from the component specifications being part of the architecture. Logical test cases can be translated into test cases at the technical level. If the logical architecture is flexible enough, it is a good starting point for working out from it units of deployment, which then can be deployed and scheduled as part of the technical architecture in terms of distributed hardware structure and its operating systems.

Artifact	Based on	Formal Description	Formal method to Work Out	Validation & Verification	Generated artifacts
Business Goals		Goal trees	Logical deduction	Logical analysis	-
Requirements	System functionality and quality model	Tables with attributes and predicate logic Taxonomies	<i>Use cases</i> Formalization in predicate logics	Consistency proof Derivation of safety assertions	System assertions System test cases
Data models	Use cases	Algebraic data types E/R diagrams Class diagrams	Axiomatization	Proof of consistency and relative completeness	-
System specification	Interface model	Syntactic interface and interface assertions Abstract state machines Interaction diagrams	Stepwise refinement	Proof of safety assertions and requirements Derivation of interaction diagrams	Interaction diagrams System test cases
Architecture	Component and composition	Hierarchy of Data flow diagrams	Decomposition	Architecture verification Architecture simulation	Interaction diagrams Integration tests
Components	Component model	Syntactic interface and interface assertions Abstract state machines	Decomposition of system specification assertions	Consistency analysis	Component tests
Implementation	State machines	State transition diagrams State transition tables	Stepwise derivation of state space and state transition rules	See component verification	
Component verification	State machine runs	Proofs in predicate logics Tests	Proof of interface assertions Test case generation	-	Test runs
Integration	Interactions	Interaction diagrams	Incremental composition	-	Test runs Interaction diagrams
System verification	Interface interaction	System interface assertions System test cases	Proof of interface assertions Test case generation	-	Test runs

**Tab. 1** Formal Artefacts, Models and Methods in Seamless Model Based Development

From the models of the architecture and its interface descriptions test cases for module tests as well as extensive integration test cases can be generated and executed. The same applies for system test.

## 5.2 Reducing Costs – Increasing Quality

One of the large potentials of formal models and techniques in development process is their effects to reduce costs. There are mainly four possibilities for cost reduction as numerated below.

1. Avoiding and finding faults early in the process,
2. Applying proven methods and techniques that are standardized and ready for use,
3. Automation of the development task and steps wherever possible,
4. Reuse of implementations, architectures requirements and development patterns wherever possible.

The last step goes into the direction of a product line engineering, which needs a highly worked out modelling and formalization approach to be able to gain all the benefits of such an approach.

## 6 Software Project Governance

One of the key success factors for the development and evolution of software-intensive systems is appropriate project governance. This includes all questions of organizing, managing, controlling and steering of software projects. A lot of insights have been gained over the years due to extensive experiences and learning from the many failures in software development. It is obvious that we need project governance that establishes cooperative processes and team organization, defines clear responsibilities for all the levels of the team hierarchies, that is able to deal with cost

estimation and meaningful reactions to cost overruns, understands the separation of work between different teams, to define responsibilities, to understand the competences needed and to install effective processes and techniques for planning, progress control, change management, version and configuration management as well as quality control.

Traditionally the two sides of technical and methodological software and system development and project governance are not well integrated. There is not much work relating management issues with issues of methods, techniques and formalisms. But this work is needed. Methods can only be effective if the management context is the right one for them and also by clever management project success cannot be enforced without effective development methods. We need much better insights into the interdependencies and the interplay between management techniques and development techniques.

## **7 Concluding Remarks: Towards a Synergy between Formal Methods and Model Based Development**

Not surprisingly the synergy between formal methods and model-based development is very deep. This synergy is not exploited in enough details so far. It is certainly not sufficient for a formal development method just to provide a formalization of informal approaches like the unified modelling language UML or to develop techniques of model checking where certain models that have been worked out in the development process. A much deeper synergy is needed where appropriate formal models directly address the structure of functionality and architecture. This concept requires targeted structures of the models and their relations by refinement. Furthermore, tracing between the models must be a built-in property supporting change requests. Then the whole model structure can be continuously updated and modified in a way such that consistent system models are guaranteed in every step of development.

Putting formal methods and modelling together we end up with a powerful concept, developing formal methods and modelling for a particular purpose, addressing particular issues in the evolution of software intensive systems with the rigour of formality and its possibilities for automation and reuse. This brings in a new quality. Achieving this, however, needs a lot of research starting from useful theories, based on valid principles, finding tractable syntax, modelling the application domain and finally integrate them into the development process and supporting it with appropriate tools.

## **References**

1. Botaschanjan, J., Broy, M., Gruler, A., Harhurin, A., Knapp, S., Kof, L., Paul, W.J., Spichkova, M.: On the correctness of upper layers of automotive systems. *Formal Asp. Comput.* 20(6), pp. 637-662 (2008)
2. Brooks, F.P. Jr.: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley (1975)

3. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer (2001)
4. Broy, M.: A Theory of System Interaction: Components, Interfaces, and Services. In: D. Goldin, S. Smolka and P. Wegner (eds.): The New Paradigm. Springer Verlag, Berlin, pp. 41-96 (2006)
5. Broy, M., Krüger, I., Meisinger, M.: A Formal Model of Services. ACM Trans. Softw. Eng. Methodol. 16(1) (February 2007)
6. Broy, M.: The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. IEEE Computer, pp. 72–80, Oktober (2006)
7. Broy, M.: Model-driven architecture-centric engineering of (embedded) software intensive systems: modelling theories and architectural milestones. Innovations Syst. Softw. Eng. 3(1), pp. 75-102 (2007)
8. Broy, M.: Interaction and Realizability, In: Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, Frantisek Plasil (eds.): SOFSEM 2007: Theory and Practice of Computer Science, Lecture Notes in Computer Science vol. 4362, pp. 29–50, Springer (2007)
9. Broy, M.: Seamless Model Driven Systems Engineering Based on Formal Models. In: Karin Breitman, Ana Cavalcanti (eds.): Formal Methods and Software Engineering. 11th International Conference on Formal Engineering Methods (ICFEM'09), Lecture Notes in Computer Science vol. 5885, pp.1-19. Springer (2009)
10. Broy, M.: Multifunctional Software Systems: Structured Modelling and Specification of Functional Requirements. Science of Computer Programming, accepted for publication
11. Hoare, T.: The verifying compiler: A grand challenge for computing research. *J. ACM* 50(1), pp. 63-69 (2003)
12. ISO DIS 26262
13. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4) (October 1992)
14. Parnas, D.L.: Some Software Engineering Principles. In: Software fundamentals: collected papers by David L. Parnas, Addison-Wesley Longman Publishing Co., Inc Boston, MA, pp. 257-266 (2001)
15. Reiter, H.: Reduktion von Integrationsproblemen für Software im Automobil durch frühzeitige Erkennung und Vermeidung von Architekturfehlern. Ph. D. Thesis, Technische Universität München, Fakultät für Informatik, forthcoming



<http://www.springer.com/978-3-642-15186-6>

The Future of Software Engineering

Nanz, S. (Ed.)

2011, VII, 185 p., Hardcover

ISBN: 978-3-642-15186-6