Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

On the Importance of Tailorable Processes in the Development of Embedded Industrial Automation Systems ⋆

Philipp Nenninger, Detlef Streitferdt

ABB Corporate Research Center, 68526 Ladenburg, Germany,
email: {philipp.nenninger | detlef.streitferdt}@de.abb.com

Abstract: Due to increasing complexity in embedded devices for industrial automation, conventional development processes put growing strain on the developers of these systems. For future generations of devices, it is therefore necessary to adapt the development process in order to be able to deliver the reliability required for these devices. In this paper, testing of embedded systems is taken as an example of how the increase in complexity drives the need for new approaches and how model-based testing can be employed to offset these effects.

1. INTRODUCTION

Although industrial automation products have a long lifetime in the field, the customer demand for new features like advanced human-machine interfaces or more recently wireless connectivity has driven an increase in complexity over the last years. In combination with traditional goals of the development for industrial automation products, namely reliability and robustness, this increase in complexity puts a large strain on the developers.

In this paper we argue that adhering to traditional development methods will in the long run lead to a decrease of product quality. We believe, that the use of tailorable development processes which are supported by a flexible and consistent tool chain can reduce the impact of the increase in complexity. Embracing new techniques like model-based testing can even lead to improved product quality despite added features and the resulting increase in complexity.

The rest of the paper is organized as follows: in Section 2, we present an overview of applicable processes, define the tailorability aspect of processes and show how model-based testing offers a solution to some of the challenges imposed on the process by increasing complexity. In Section 3 we present our vision of a flexible tool chain and compare it to the state-of-the-art. The paper ends with a conclusion, which presents some challenges for the parties involved in the development of intelligent devices in industrial automation.

2. DEVELOPMENT PROCESS

Typical and intuitive examples for processes used in the development of embedded systems are the waterfall model and the V-Model, see Bunse and von Knethen (2002). Both provide engineers with a guideline of how to develop an embedded system. In practical development however, following defined guidelines and processes and even the correct and complete implementation of development processes is a challenge. This is true especially in larger projects. In this section, we will look at the reasons and consequences for this and use model-based testing as an example of how to effectively use methods to improve product development. It is thus an improvement of the way towards a product. Of course, as result of this, the quality of the product will increase.

2.1 Understanding how development works

Most development processes commonly share three fundamental phases of system development: understanding what to develop (1), development of the product (2) and checking whether the outcome matches what was originally intended (3). Which phase has to be repeated at what frequency and how many iterations are required depends on the system under consideration and the process used. Processes are defined as hierarchical set of process steps.

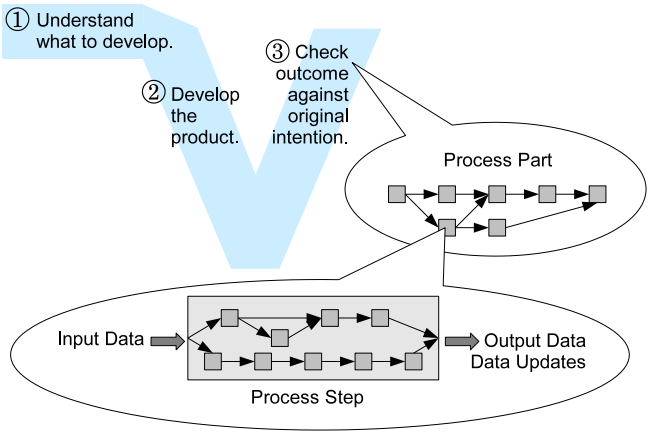

Fig. 1. Process step in the context of a process, see Streitferdt and Nenninger (2007)

In Fig. 1 the three fundamental phases are depicted as part of the V-Model, which is an abstraction chosen for this paper. Each phase is then extended to a set of process steps, which in turn are part of the complete development

⋆ This work was supported in part by the European Union ITEA2 project D-MINT.

process – a process part. Process steps have an input as well as an output, which is the data that is processed. This data can be plain or formatted text, mathematical formulas or even an UML model. Note, there may be more than one way through a process part and still each way has to be correct. These differences come from the various methodologies that can be used or are in use. Finally, each process step is made up of several actions. Again, there are many possible ways to convert the input data into the output data. The variability of a process step mainly comes from cultural peculiarities. All in all we get to an adaptable/tailorable process. The principle of arranging and tailoring processes is well known and implemented e.g. in the freely available Eclipse Process Framework [1] (EPF).

None of these steps is trivial even in isolation. Understanding what to build for example includes understanding the customer's needs which can be challenging even between engineers and computer scientists. The main challenge in the requirements phase is the multi-domain knowledge which is needed on both sides, that of the stakeholders and that of the implementing engineers. Experience shows that defining crucial vocabulary like "system" and "architecture" leads to more productive discussions and thus to better requirements, which are the main work product of this process step. Good requirements have three properties: they are correct, concise and complete. If the requirements lack any of these properties, the chances that there will be large changes late in the development process are high. For the formal part of requirements engineering, there are several good sources, for example Ardis (1997), Atlee et al. (1996), and Chechik and Gannon (1994). However the challenge of transferring domain knowledge to a formalized representation remains. Gathering, maintaining and re-using best practice knowledge is a key success factor in developing embedded systems.

During the implementation phase of the development process ("(2) Develop the product"), working with large teams poses another set of challenges which can be aggregated under "corporate culture", see also Kankanhalli et al. (2002). Teams do not have to work on different continents for differences in corporate culture to appear, in some cases two departments or teams at different sites have different views of how development should work. One example is the level of detail engineers expect from the documents they receive, which require a delicate balance between over-specification, resulting in engineers who feel that their creativity is not valued, and under-specifying the problem leaving engineers to wonder what to develop in the first place. In order to account for varying corporate cultures among other things, a process has to be *tailorable*. Tailoring adapts a process to the needs of the project under consideration by adding or removing process steps and work products. In the example given above, this implies adding more detailed requirements and models if the developers feel that they are not given sufficient information to complete their tasks. Tailorability is a crucial feature of development processes because it allows the process to be fitted to varying environments, products and tools.

One example of such a tailorable process is MeDUSA [2], see Nyssen and Lichter (2007), which was developed in a cooperation between the ABB Corporate Research Center Germany and the RWTH Aachen and is extensively used for example in the development process of industrial instrumentation products at ABB. MeDUSA is based on the COMET method Gomaa (2000) and has been tailored to be object-based instead of object-oriented since this allows for a smoother transition between models and C-code. Object-based means that some features of object-oriented languages like polymorphism are not or only partially covered. MeDUSA covers the development process from early use-cases to the detailed design by using the UML2 notation.

In the development process of embedded systems, testing is an important phase. Depending on the size of the system under test and the reliability requirements, between 30 % and 50 % of the total development time are devoted to testing. Embedded system testing requires an effort to develop test-cases which can be based on or even generated from a test model. There are various notations for such a test model, a very common one is the finite state machine (FSM). For reasons explained in detail in the following section, testing in its conventional form can be seen as a key factor in limiting the ability to increase the complexity of industrial automation products. It therefore has the potential to become a driving factor in embracing new technologies and processes.

2.2 The limits of growth

One popular and simple transition-based modeling technique is the use of finite state machines (FSMs), which use nodes for the states and directed vertices for the transitions of system. The usage or a test of the system can be described as a (not necessarily finite) path through the FSM. In a first and very rough approach, an FSM of an embedded system can be characterized as a tree with its root at the reset state and the leaves being the states at the end of an execution. Every independent decision, that is every if-statement, creates a new level and thus the number of states x. With n being the number of (binary) variables which are checked in the if-statements, the number of states results to

$$x = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

Because the length d of a path from the root to a leaf is n, the combined length of all paths from the root to the leaves

$$\sum d = n \cdot 2^n$$

and thus also grows faster than exponentially with n.

While this model might not be very accurate, it neglects shorter paths and joining paths for example, it describes in simple math a phenomenon often observed by developers:

[1] See `www.eclipse.org/epf` for further details.

[2] Method for Designing UML2-based Embedded System Software Architectures

that even small increases in functionality can result in a large increase in complexity and thus testing effort [3].

It becomes clear that complete path or state coverage becomes increasingly difficult with project size just by the sheer magnitude of the system. With test engineer executing each test case manually, the cost of testing soon becomes prohibitive. Additionally, if the process to generate the test cases is procedure-free, that is if test cases are just 'guessed', determining the state or path coverage is another challenge.

This short example motivates the statement made above that industrial-strength testing is indeed a limiting factor for the complexity of systems. But if customers require additional features which drive complexity and traditional testing cannot deliver, another approach to the problem is needed.

2.3 Model-based testing

In the past few years, the formal verification of software [4] has grown by leaps and bounds, see for example Chaki et al. (2004) and Ray and Sumners (2007). For specialized software, it has even reached a level of maturity where it can be considered to be state-of-the-art. A good example is the Static Driver Verification for Windows, coming out of Microsoft's SLAM project, see Ball et al. (2006). Due to its (relatively) narrow scope, it checks for application programming interface (API) usage errors of device drivers, the construction of a model for the environment, in this case the operating system and some other pieces of software, was possible.

For embedded systems in general and even for embedded systems in industrial automation, the development of an environment model to cover external influences on the system is currently only possible for parts of the environment. Here, static code analysis has reached a level of maturity that allows it to be used productively during the implementation phase. In static code analysis, the source code of a system is checked for several typical programming faults such as boundary violations but also more complex problems like the real-time capability of the embedded system. Tools, which operate directly on the source code are readily available. But testing is still by far the most important way to determine, whether the system fulfills its requirements. When it comes to testing, two central questions have to be answered:

What to test against: Before testing can take place, the test team has to have a reference, a test oracle, to be able to decide, whether the system under test passed or failed a specific test.

How to test: Test strategies vary from black to white box testing. Which of these strategies is applicable depends on the system under test and can even differ for various parts of the same system. In general, exhaustive white-box testing (testing with all possible permutations of input variables) is impracticable, instead equivalence classes are in use.

On both of these topics there are many excellent books which describe in detail what the theoretical options of the test team are and how to decide on the best test for a given system, see Myers (2004) for example. In practice, however, the nature and the amount of tests is limited by the effort involved with testing. With a given set of test cases the corresponding execution effort can be calculated, however the number of test cases only has a proportional influence on the effort of the testing process, however, certain parts of the system which are covered by the new test cases have already been tested in existing test cases and therefore an increase in the number of test cases does not translate directly into an increased coverage, as indicated in Fig. 2. The coverage here is the ratio of a certain metric which is covered by the test cases. Popular choices for the coverage metric include branches, decisions and states. As illustrated by the example above, the first test cases usually yield a much larger increase in coverage than the subsequent ones, and the return on investment diminishes rather quickly for conventional testing. This is especially true if regression testing of the system is taken into consideration. For all practical purposes, the coverage converges towards an upper limit.

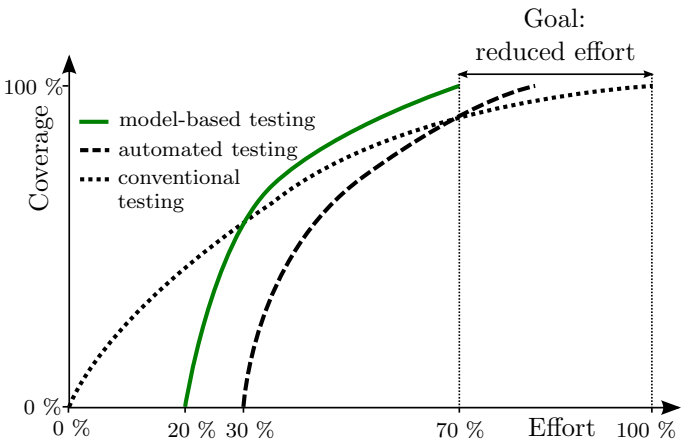

Fig. 2. Estimated effort for a given coverage using different testing approaches. Break-even points, upper limits and reduced effort vary depending on the project under consideration.

An attribute of "conventional testing" is that both the test case definition and the test case execution are done manually. This form of testing is still common (and effective) for small subsystems which will not be included in the final product. However, this form of testing quickly becomes inefficient as the system under test gets larger.

One common way to increase coverage during testing is the use of automated testing, for which tools and processes are available. In contrast to conventional testing, tests can run unsupervised in batch mode and thus much more efficiently in most cases. A common example of automated testing are hardware-in-the-loop (HiL) test rigs which are widely used in the automotive industry. Automated testing requires a certain investment before the start of the tests, which is indicated in Fig. 2 as an offset on the "effort"-axis. This investment comes in the form of installing the more complicated test rig and translating the test cases,

[3] It is worthwhile mentioning that Uttig and Legeard (2007) come to the same statement 'exponential growth of effort' by using an entirely different approach.

[4] The term 'formal verification of software' may be a bit misleading here since it is really checked to see if certain assumptions about the piece of software under consideration hold under certain conditions.

so they can be executed automatically, for example from an informal list to a formalism such as TTCN-3, see for example Wilcock et al. (2005).

The break-even point between conventional and automated testing depends on many factors including the type of system under test, the experience of the test engineers and of course the coverage metric applied. If for example the test setup and the test cases of a previous product can be reused, and the test team is already experienced in using the automated testing environment, the break-even point shifts in favor of automated testing.

The question "what to test against" is usually answered rather quickly: test cases come with expected results. But that is only part of the answer. What is still unclear is how these test cases are generated and how the outcome is evaluated. One of the few cases where predicting the outcome is not a problem is the re-engineering of a system due to a changed bill of materials but with unchanged functionality. In most cases however, an oracle in the form of a technical system is not available and predicted test results have to be explicitly generated. Some authors argue that this process always requires a *model* of the system, even if this model exists only in the test engineer's head, for example Uttig and Legeard (2007). While there is some truth in this point of view, we would like to call this an *informal model* and distinguish it from the more formal models, which we will refer to simply as *models*. Fig. 3 shows a comparison of a simple implementation process and a concurrent test process. We would like to focus on the testing side of the diagram and thus, the implementation side is not shown in detail.

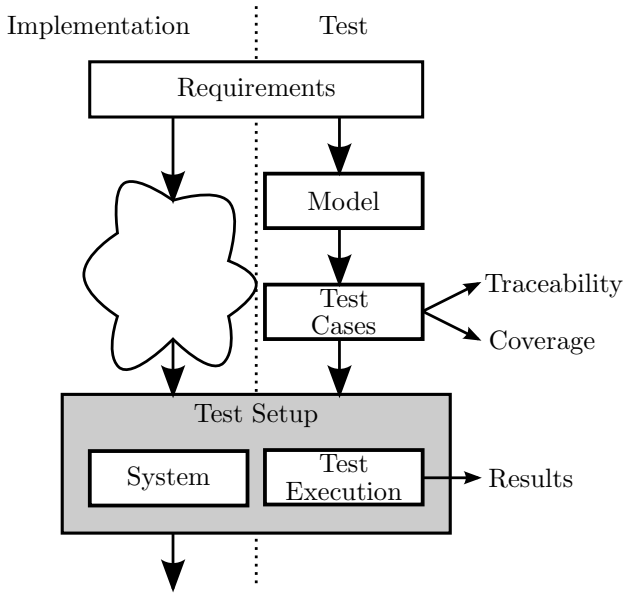

Fig. 3. Model-based testing as a parallel activity to system implementation

An important point in Fig. 3 is, that both the implementation and the testing branch have their root in the requirements. In Subsection 2.1 we stressed the importance of correct, concise and complete requirements. Since both the system and the test cases are developed based on the requirements, a fault in the requirements will lead to faulty behavior of the system under test which will not be evaluated as a failed test and can therefore not be detected using model-based testing.

In Fig. 3, we chose to create a model specifically for testing. Since we did not specify the implementation activities, it is possible, and with a process that includes model-based testing it is even likely, that development is also model-based. This raises the question, whether or not development and test should share a common model. On one hand, reusing the development model for test design saves effort and thus money, avoiding 'redundant' work. On the other hand, auto-generating the complete code and all test cases for the system from the same model basically limits the testing results to a statement about differences in the development and the testing tool chain. Although this can be useful, it does not cover implementation faults. A crucial question is therefore, when to split the two paths. The aim of having a model for model-based development is to include as much detail as possible for example in the form of state machines, because this information is required for the code generation. Model-based testing does not require this level of detail, since white-box testing of individual building blocks, classes or functions, is covered by unit testing. If the development path produces a high level model, this model can be used to generate test cases. However it must be possible to verify that the model at this abstraction level satisfies the requirements.

After generating the test cases from the model, the quality of these test cases can be evaluated. Typically, the *coverage* is used as a metric. Depending on the system under test, various coverage metrics, for example decision, path or code coverage, can be applied to evaluate the efficiency of the test cases. It is also possible to assess the relevance of test cases based on the typical use of the system. Which metric is used strongly depends on the system under test but the metric gives the engineer some idea whether the testing effort is sufficient, if new test cases must be introduced or if a subset of the test cases created from the model can be selected in order to fulfill other prerequisites, such as limited time for overnight testing, and still achieve a good coverage. Another important feature of the test cases is their traceability to the requirements, usually given in the form of a traceability matrix linking test cases and requirements in a many-to-many relation.

The coverage matrix is one point where it is possible to track, whether the different pieces of the process puzzle come together. Traceability is an important metric in itself, since it allows to measure whether or not, and even to what degree the test cases consider certain requirements. If the model includes usage or cost information the traceability matrix also provides information about how intensively the most probable and the cheapest or most expensive requirements are represented in the test cases. Based on this information, the set of test cases can be refined in order to meet testing goals like minimum (statistical) coverage or test duration.

In comparison to conventional testing, model-based testing has two major advantages:

(1) Better than with conventional testing, several coverage metrics can be used for the evaluation of the test cases. In case special properties must be covered, for example special paths, test cases can be specifically

designed. This allows for an effective approach to test case design.

(2) Since the execution of the test cases is automated, generating and executing more test cases from a given model does not increase the time required from the test team (proportionally), but only the time the test rig is occupied. Therefore, testing is possible earlier and more often in the development process. It is also easier to re-test the system after parts of the system have been updated.

The quality of the final product is of course related to the way it was developed. The more structured and clearer a development process is, the better the resulting products will be. In the process described above, we started with the requirements, the design of the system and the resulting source code. With the given testing methods, strategies, technologies and the current complexity of embedded systems it became clear that test cases have to be generated out of a test model to reach the best possible effort reduction. Clearly the only way towards model-based testing will lead via a well defined and tailorable development process. Tailorable, because the current situation has a level of diversity which simply prohibits a one-fits-all-solution, with the given effort reduction goal.

The fact that the test execution is automated requires some sort of special tools for this task. How this and other tools should work together to effectively support development is described in following section.

3. TOOL CHAIN

For the whole process described in Section 2, commercial as well as, for some parts, open source tools are available. Use-cases can be modeled using standard UML tools, requirements can be written using a text processor and source code can be processed by command line tools like compilers/debuggers. It is possible to deliver high quality products with this tool-setup, but it is not possible to deliver the currently needed product quality within the currently required time-to-market for currently needed systems. The 'old' way of developing embedded systems described above will work well for very small systems with less than about some thousand lines of code. For the rest, more than 90 %, of the embedded systems structured, defined and accepted development processes are a prerequisite for current high quality products. Still the goal of a tailorable tool chain–process combination has not been reached sufficiently.

The usage of a development process with its associated tools for a process step currently requires the manual transformation of the results of one process step into an input for the next step. This transformation is error-prone and requires additional effort of system developers. Also, it is occasionally omitted to track changes late in the development, which are directly implemented in source code, in the results of the early stages of the process, the requirements for example.

Several tool vendors have put considerable effort into completing their tool chain, Telelogic [5] and The Mathworks [6] being just two examples. These tool chains provide good support for their specific domain, but tend to lock the developer in, which is not just a financial issue, but also prevents him to use the 'right tool for the job'. The reasons for the tools selections are closely linked to those for process tailoring, see Section 2. Just like the tailorability of processes, the ability to exchange unfitting tools is a crucial feature of a tool chain for the development of complex industrial automation products. Using the import/export features of tools is often insufficient since vendors tend to focus on the import feature and neglect the export, leading to data loss in the transformation. In Streitferdt and Nenninger (2007), we proposed the use of data (meta)-models, like the one provided by Eclipse [7] as a possible solution for this problem, see also Figure 4.

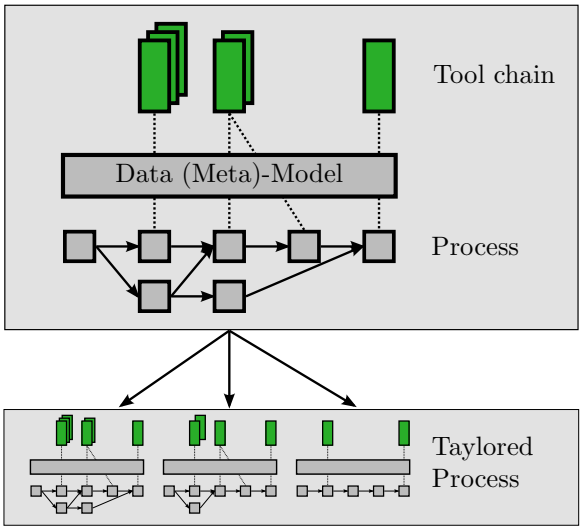

Fig. 4. Tailoring of a process

The data (meta)-model contains all the information created in the process and thus links the process steps. The different tools are associated to one or more specific process steps and several tools may be used in one process step. Since they all store their output in the data (meta)-model, they are exchangeable and developers can chose the tool they feel is appropriate without the danger of losing information or creating results incompatible with those of other tools. The data (meta)-model is the general idea of how to integrate various tools into a meaningful tool-chain. The model-bus is an approach going more towards keeping the tool boundaries, see e.g. Blanc et al. (2005). With pre-defined transformations for a data exchange (a partial data exchange is also allowed) between tools it offers a possibility to arbitrarily interconnect tools and establish a user-definable tool chain. However these transformations typically include some sort of data loss due to varying representation of the same data in different tools or even differing data sets.

From the viewpoint of the developers of embedded systems in the industrial automation domain, the convergence of tools toward a common data (meta)-model is a desirable developments, possibly comparable to the rise of embedded real-time operating systems. Like using a real-time op-

[5] www.telelogic.com
[6] www.mathworks.com
[7] See www.eclipse.org for more information on the Eclipse data model.

erating system, having a continuous and tailorable development process with related tool chain lets the developer focus on the problem at hand instead of using up effort on process or tool integration questions. A continuous tool chain in combination with a tailored process will in any case ease solving the traceability challenge to, e.g. track errors found in the integration testing back to all the elements involved (requirements, model elements, source code). Whether or not it is feasible to do this automatically remains to be seen.

4. CONCLUSION AND OUTLOOK

Development processes became more and more important for embedded systems which have been considered to be very small and easy to handle. The complexity increase of these systems now requires an adapted approach in the development process – tool chain question. The diversity of the embedded systems in the industrial automation domain needs highly tailorable processes as well as very flexible tools.

As shown in the testing of embedded systems example, reliability and robustness, traditional goals of the development for industrial automation products, require state-of-the-art technology adaptation. Model-based testing as such a technology requires its integration into development processes already in place. Additionally, the tools needed to realize and efficiently run a model-based testing approach have to be integrated into an existing tool chain. Of course all this has to happen without disrupting the normal business and the product release schedules.

A flexible process base with integrated tool chain is a general solution to this challenge, which has been presented for the industrial automation domain. Future efforts will be spent on realizing this approach and pushing forward current solution ideas in this field.

REFERENCES

Mark A. Ardis. *Formal Methods for Telecommunication System Requirements: A Survey of Standardized Languages*. Bell Laboratories, 1997.

Joanne Atlee, Marsha Chechik, and John Gannon. *Using Model Checking to Analyze Requirements and Design*. Department of Computer Science, University of Waterloo, 1996.

T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys conference*, pages 73–85. ACM Press New York, NY, USA, 2006.

X. Blanc, M.P. Gervais, and P. Sriplakich. Model Bus: Towards the interoperability of modelling tools. *MDAFA*, 4:10–11, 2005.

Christian Bunse and Antje von Knethen. *Vorgehensmodelle kompakt*. Spektrum Akademischer Verlag, 2002.

S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

Marsha Chechik and John Gannon. Automatic verification of requirements implementation. In *Proceedings of the 1994 Internation Symposium on Software Testing and Analysis (ISSTA)*, 1994.

Hassan Gomaa. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley, 2000.

Atreyi Kankanhalli, Bernard C.Y. Tan, Kwok-Kee Wei, and Monica C. Holmes. *Cross-Cultural Differences and Information Systems Developer Values*. Department of Information Systems, School of Computing, National University of Singapore, 2002.

Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, 2. edition, 2004.

Alexander Nyssen and Horst Lichter. MeDUSA – method for UML2-based design of embedded software applications. Technical report, RWTH Aachen – Department of Computer Science, 2007.

S. Ray and R. Sumners. Combining Theorem Proving with Model Checking through Predicate Abstraction. *IEEE Design & Test*, 24(2):132–139, 2007.

Detlef Streitferdt and Philipp Nenninger. Quality assurance challenges in the industrial automation domain. In *Proceedings of CONQUEST*, 2007.

Mark Uttig and Bruno Legeard. *Practical Model-Based Testing*. Morgan Kaufmann Publishers, 2007.

Colin Wilcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiley and Sons, 2005.